Object Oriented Programming Notes - Last Minute Revision

Most Asked OOPS Interview Questions

1: What is OBJECT-ORIENTED PROGRAMMING?

Answer: Object-oriented programming is a programming paradigm built on the concept of objects.

In Other Words, it is an approach to problem-solving where all computations are carried out using objects.

2: Class and Object

Answer:

Class: A class is the building block that leads to Object-Oriented programming. It is a user-defined datatype, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Object: An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
#include <bits/stdc++.h>
using namespace std;
class Person{
        // Access specifier
        public:
// Data Members
        string name;
// Member Functions()
        void printname(){
          cout << "Person name is: " << name;
        }
};
int main() {
// Declare an object of class Person
        Person obj1;
// accessing data member
        obj1.name = "Thanos";
// accessing member function
        obj1.printname();
        return 0;
}
```

Output

```
Person name is: Thanos
```

3: Constructor

Answer:

- Constructors are special class members which are called by the compiler every time an object of that class is instantiated.
- Constructors have the same name as the class and may be defined inside or

outside the class definition.

There are 3 types of constructors:

1. Default constructors
2. Parameterized constructors
3. Copy constructors

1. **Default Constructor**: Default constructor is the constructor which doesn't take any argument. It has no parameters.

2. **Parameterized Constructor**: A constructor is called Parameterized Constructor when it accepts a specific number of parameters.

3. **Copy Constructor**: A copy constructor is a member function which initializes an object using another object of the same class.

Characteristics of the constructor:

- Constructor has the same name as the class itself.
- Constructors don't have a return type.
- A constructor is automatically called when an object is created.
- It must be placed in the public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).
- Constructors can be overloaded.
- Constructor cannot be declared virtual.

```
#include <bits/stdc++.h>
using namespace std;
class student
{
     string name;
     public:
       int age;
       bool gender;

    // Default Constructor
    student(){
      cout<<"Default Constructor"<<endl;
    }

    // Parameterised Constructor
    student(string s, int a, int b)
    {
       name = s;
       age = a;
       gender = b;
       cout <<"parameterised constructor"<<endl;
    }
// Copy Constructor
    student (student &p){
      name = p.name;
      age = p.age;
      gender = p.gender;
      cout<<"copy constructer"<<endl;
    }
void printinfo()
    {
        cout << "Name = ";
        cout << name << endl;
        cout << "Age = ";
        cout << age << endl;
        cout << "Gender = ";
        cout << gender << endl;
        cout << "\n";
    }
};
int main()
{
    // Default Constructer Call
    student s1;
    s1.printinfo();
    // Parameterised Constructer Call
    student s2("sumeet", 20, 1);
    s2.printinfo();
    // Copy Constructor Call
    student s3(s2);
    s3.printinfo();
return 0;
}
```

Output

```
Default Constructor
Name =
Age = 2
Gender = 0
parameterised constructor
Name = sumeet
Age = 20
Gender = 1
copy constructer
Name = sumeet
Age = 20
Gender = 1
```

4: Destructor

Answer:

- A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor.

- Destructor has the same name as their class name preceded by a tiled (~) symbol.

Characteristics of the constructor:

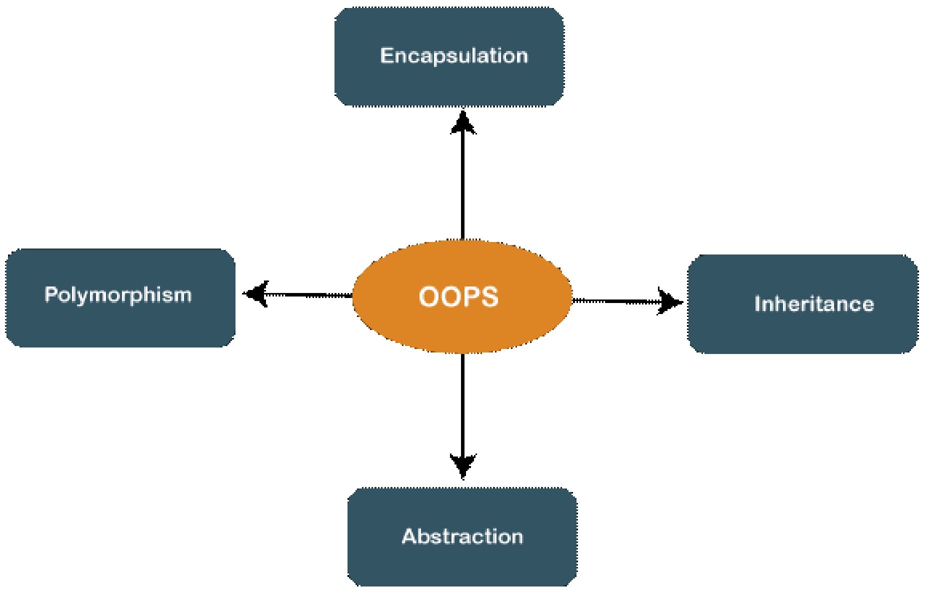
- Destructor is invoked automatically by the compiler when its corresponding constructor goes out of scope and releases the memory space that is no longer required by the program.
- Destructor neither requires any argument nor returns any value therefore it cannot be overloaded.
- Destructor cannot be declared as static and const.
- Destructor should be declared in the public section of the program.

```
#include <iostream>
using namespace std;
int count = 0 ;
class num{
public:
    num(){ // Constructor
        count++;
        cout << "This is the time when constructor is called for object number" << count << endl;
    }
~num(){ // Destructor
```

```
        cout << "This is the time when my destructor is called for object number" << count << endl;
        count--;
    }
};
```

5: The main features of OOPs?

Answer: The main four pillar of oops are given below.

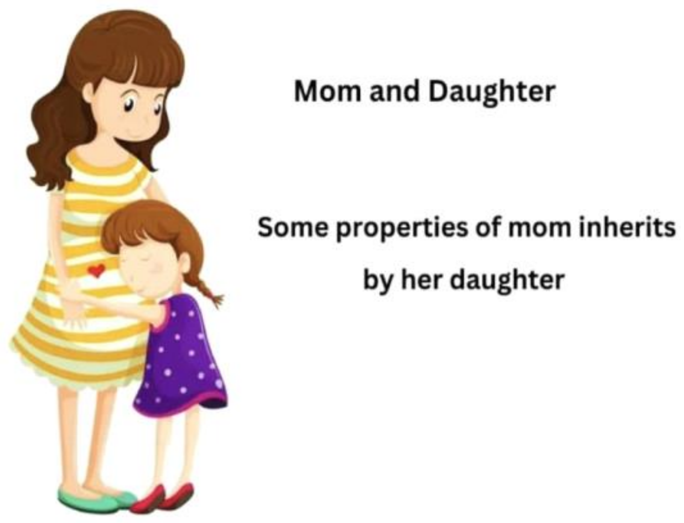


6: Inheritance

Answer: Inheritance is one of the most important features of Object-Oriented Programming. The capability of a class to derive properties and characteristics from another class is called Inheritance.

Real Life Example

Inheritance



There are 5 types of Inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance.
5. Hybrid Inheritance.

1. **Single Inheritance**: When a subclass(child) is inherited from a base class is called single inheritance.

```
#include<bits/stdc++.h>
using namespace std;
class A{
    public:
      void funcA(){
        cout<<"Base Class"<<endl;
      }
};
// Class B is inherited from Class A
class B : public A{
    public:
      void funcB(){
        cout<<"Inherited from class A"<<endl;
      }
};
int main(){
    B obj;
    // As Class B inherited properties of A.
    // We can access funcA from class B object also.
    obj.funcA();
    obj.funcB();
    return 0;
}
```

Output

```
Base Class
Inherited from class A
```

2. **Multiple Inheritance**: when one subclass is inherited from more than one base class is called multiple inheritance.

```
#include<bits/stdc++.h>
```

```
using namespace std;
class A{
    public:
      void func(){
        cout<<"Base class A"<<endl;
      }
};
class B{
    public:
      void func(){
        cout<<"Base class B"<<endl;
      }
};
// Class C inherits both Class A and B
class C : public A, public B{
public:
      void func(){
        cout<<"Inherited from class C"<<endl;
      }
};
int main()
{
    C obj;
    obj.A :: func();  // resolving ambiguity
    obj.B :: func();
    obj.func();
    return 0;
}
```

Output

```
Base class A
Base class B
Inherited from class A and B
```

3. **Multilevel Inheritance**: In this type of inheritance, a derived class is created from another derived class.

```
#include<bits/stdc++.h>
using namespace std;
class A{
     public:
       void funcA(){
         cout<<"Base class A"<<endl;
       }
};
// Class B inherited from Class A
class B : public A{
     public:
       void funcB() {
         cout<<"Inherted from class A"<<endl;
       }
};
// Class C inherited from Class B
class C : public B{
     public:
       void func() {
         cout<<"Inherited from class B"<<endl;
       }
};
int main()
{
    C obj;
    obj.funcA();
    obj.funcB();
    obj.func();
    return 0;
}
```

Output

```
Base class A
Inherted from class A
Inherited from class B
```

4. **Hierarchical Inheritance**: In this type of inheritance, more than one subclass is inherited from a single base class.

```
#include<bits/stdc++.h>
using namespace std;
class A{
    public:
      void funcA(){
        cout<<"Base class A"<<endl;
      }
};
// Class B inherited from Class A
class B : public A{
    public:
      void funcB(){
        cout<<"Inherited from class A"<<endl;
      }
};
// Class C also inherited from Class A
class C : public A{
    public:
      void funcC(){
        cout<<"Inherited also from class A"<<endl;
      }
};
int main()
{
    C obj;
    obj.funcA();
    obj.funcC();

    B obj2;
    obj2.funcA();
    obj2.funcB();
    return 0;
}
```

Output

```
Base class A
Inherited also from class A
Base class A
Inherited from class A
```

5. **Hybrid Inheritance**: The inheritance in which the derivation of a class involves more than one form of any inheritance is called hybrid inheritance. Basically C++ hybrid inheritance is combination of two or more types of inheritance. It can also be called multi path inheritance.

```
#include <iostream>
using namespace std;
class A
{
            public:
              int x;
};
class B : public A
{
            public:
              //constructor to initialize x in base class A
              B()
              {
                 x = 10;
              }
};
```

```
class C
 {
            public:
              int y;

              //constructor to initialize y
              C()
              {
                  y = 4;
          }
};
//D is derived from class B and class C
class D : public B, public C
{
            public:
              void sum()
              {
                  cout << "Sum = " << x + y;
              }
};
int main()
{
            //object of derived class D
        D obj1;
            obj1.sum();
            return 0;
}
```

Output

```
Sum = 14
```

7: Encapsulation

Answer:

- In normal term encapsulation is defined as wrapping up of data and information under a single unit.
- Encapsulation define as binding together the data and function that manipulates them.

Advantages

- Increased security of data.
- Encapsulation allows access to a level without revealing the complex details below that level.
- It reduces human errors.
- Makes the application easier to understand.

Real Life Example

Encapsulation
School bag can keep your book, pen, erasers, lunch box so on ...

```
#include<iostream>
using namespace std;
class Encapsulation
{
        private:
          // data hidden from outside world
          int x;

        public:
          // function to set value of
          // variable x
          void set(int a)
          {
                x =a;
          }

          // function to return value of
          // variable x
          int get()
          {
                return x;
          }
};
int main()
{
        Encapsulation obj;

        obj.set(5);

        cout<<obj.get();
        return 0;
}
```

Output

```
5
```

8: Abstraction

Answer:

- Data Abstraction is one of the most essential and important feature of Object Oriented Programming in c++.
- Abstraction means displays only the relevant attributes of objects and hides the unnecessary details like the background details and implementation.

Advantages

- Helps user to avoid writing the low level code.
- Avoids code duplication and increases reusability.
- Helps to increase security of an application or program as only required details are provided to the user.

Real Life Example

Abstraction
Check balance
Deposit cash
Withdrawal cash
Print bill
@iamrupnath
Even though it performs a lot of actions it doesn't show us the process. It has hidden its process by showing only the main things like getting inputs and giving the output.

```
#include <iostream>
using namespace std;
class implementAbstraction {
private:
   int a, b;
public:
   // method to set values of
   // private members
   void set(int x, int y)
   {
            a = x;
            b = y;
   }
void display()
   {
            cout << "a = " << a << endl;
            cout << "b = " << b << endl;
   }
};
int main()
{
            implementAbstraction obj;
            obj.set(10, 20);
            obj.display();
            return 0;
}
```
Output

```
a = 10
b = 20
```

9: Polymorphism

Answer:

- The word polymorphism means having many forms. Polymorphism occurs when there is a hierarchical mode inheritance.
- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Advantages

Polymorphism in C++ allows us to reuse code by creating one function that’s usable for multiple uses. We can also make operators polymorphic and use them to add not only numbers but also combine strings. This saves time and allows for a more streamlined program.

Real Life Example

Polymorphism
In school behave like a student
In shopping mall behave like a customer
In home behave like a son
In bus behave like a passenger

There are 2 types of Polymorphism:

1. Compile time Polymorphism
2. Run time Polymorphism

1. **Compile time Polymorphism**: Compile-time polymorphism is a polymorphism that is, the function call is resolved during the compilation process.

We can achieve Compile-time polymorphism by two ways:

1. **Function overloading**

- When there are multiple functions with the same name but take different parameters as an arguments then these function are said to be overloaded.
- Functions can be overloaded by changing the number of arguments or and

changing the type of arguments.

```
#include<bits/stdc++.h>
using namespace std;
class Simple
{
  public :
    void fun()
    {
        cout<<"function with no argument"<<endl;
    }
void fun(int x)
    {
        cout <<"function with int argument"<<endl;
    }
    void fun(double x)
    {
        cout<<"function with double argument"<<endl;
    }
};
int main()
{
    Simple obj;
    obj.fun();
    obj.fun(4);
    obj.fun(4.5);
    return 0;
}
```

Output

```
function with no argument
function with int argument
function with double argument
```

2. **Operator Overloading**: C++ also provides the option to overload operators So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

```
#include<iostream>
using namespace std;
class Complex {
private:
   int real, imag;
public:
   Complex(int r = 0, int i = 0) {real = r; imag = i;}

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator + (Complex const &obj) {
                Complex res;
                res.real = real + obj.real;
                res.imag = imag + obj.imag;
                return res;
   }
   void print() { cout << real << " + i" << imag << '\n'; }
};
int main()
{
             Complex c1(10, 5), c2(2, 4);
             Complex c3 = c1 + c2;
             c3.print();
}
```

Output

```
12 + i9
```

2. Runtime Polymorphism

- Runtime polymorphism is also known as dynamic polymorphism or late binding. In runtime polymorphism, the function call is resolved at run time.

- This type of polymorphism is achieved by Function Overriding or Virtual function.

Virtual Function

- Virtual is a keyword in C++
- A virtual function is a member function in the base class that we expect to redefine in derived classes
- When a virtual function is defined in a base class, then in runtime on the basis of type of object assigned to it, the respective class function is called.

```
#include <iostream>
using namespace std;
class Base {
   public:
    virtual void print() {
        cout << "Base Class Function" << endl;
    }
};
class Derived : public Base {
   public:
    void print() {
        cout << "Derived Class Function" << endl;
    }
};
int main() {
    // Create the pointer of base class
    Base* bptr;
    // Create the object of base and derived class
    Base base;
    Derived derived;

    // In runtime, its depend on which class object
    // we are assigning in base pointer.

    // Base Class print function will call,
    // as we assign base class object
    bptr = &base;
    bptr->print();

    // Derived Class print function will call,
    // as we assign derived class object
    bptr = &derived;
    bptr->print();
return 0;
}
```

Output

```
Base Class Function
Derived Class Function
```

10: Abstract Class

Answer

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class. 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().
- Class is Abstract, if we have atleast one pure virtual function.

11: Pure Virtual Function

Answer

- Also called Absract function.
- A pure virtual function in c++, is a virtual function for which we can have implementation, but we must override that function in the derived class, otherwise the derived class will also become abstract class.

```
class X
{
        public:
        virtual void show() = 0; // pure virtual func
};
```

12: Friend Class & Friend Function

Friend Class

Answer

- A friend class can access private and protected members of other class in which it is declared as friend.
- It is sometimes useful to allow a particular class to access private members of other class.

```
#include<iostream>
using namespace std;
class A{
   int x;
   public:

    A(){
      x=10;
    }
    friend class B; //friend class
};
class B{
    public:
      void display(A &t){
             cout<<endl<<"The value of x="<<t.x;
      }
};
int main(){
        A _a;
        B _b;
        _b.display(_a);
        return 0;
}
```

Output

```
The value of x=10
```

Friend Function

Answer

- Like a friend class, a friend function can be granted special access to private and protected members of a class in C++.
- They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends.

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

class Base {
```

```
private:
    int a;

protected:
    int b

public:
    Base()
    {
        a = 1;
        b = 2;
    }

     // friend function declaration
    friend void func(base& obj);
};


// friend function definition
void func(base& obj)
{
    cout << "Private Variable: " << obj.a << endl;
    cout << "Protected Variable: " << obj.b;
}

// driver code
int main()
{
    Base obj;
    funcobj);

    return 0;

 }
```

Output

```
Private Variable: 1
Protected Variable: 2
```

13: Access Modifiers

- **Private** – The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default** – The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected** – The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public** – The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

From <https://github.com/aman0046/LastMinuteRevision-OOP>