OOPS

It is basically a programming style that uses the concept of class and object in programming. The popular object-oriented programming languages are c++, java, python, PHP, c#, etc. The main objective of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc.

Class

A user-defined type that describes what a particular kind of object will look like. Thus, a class is a template or blueprint for an object. A class contains variables, methods, and constructors.

```
class class_name{
    // properties
    // methods
};
```

Methods - Functions inside class.

Constructor

Constructors are special class functionsthat perform the initialization of every object. In C++, the constructor is automatically called when an object is created. It is a special method of the class because it does not have any return type. It has the same name as the class itself. Once we create any constructor manually default constructor get destroyed.
There are three types of constructors in C++:
★ Default constructor
★ Parameterized Constructor - takes the arguments
★ Copy Constructor - These are a particular type of constructor that takes an object as an argument and copies values of one object’s data members into another object. We pass the class object into another object of the same class in this constructor. As the name suggests, you Copy means to copy the values of one Object into another Object of Class. This is used for Copying the values of a class object into another object of a class, so we call them Copy constructor and for copying the values.

```
#include <bits/stdc++.h>
using namespace std;

class smartphone{
private:
    // Data Members(Properties)
    string model;
    int year_of_manufacture;
    bool _5g_supported;

public:
    // default constructor
    smartphone()
    {
        model = "unknown";

        year_of_manufacture = 0;
        _5g_supported = false;
    }
    // parameterized constructor
    smartphone(string model_string, int manufacture, bool _5g_)
    {
        // initialising data members
        model = model_string;
        year_of_manufacture = manufacture;
        _5g_supported = _5g_;
    }
    // copy constructor
    smartphone(smartphone &obj)
    {
        // copies data of the obj parameter
        model = obj.model;
        year_of_manufacture = obj.year_of_manufacture;
        _5g_supported = obj._5g_supported;
    }
};

int main()
{
    // creating objects of smartphone class

    //  using default constructor
    smartphone unknown;
    //  using parameterized constructor
    smartphone iphone("iphone 11", 2019, false);
    // using copy constructor
    smartphone iphone_2(iphone);
    return 0;
}
```

why do we use private constructor?

1. Singleton class - only one object or instance can be created.
2. Prevent Inheritance - By declaring the constructor as private, you prevent the creation of derived classes since derived classes need access to the base class constructor.
3. Non-instantiable classes - There are cases where you might create a class that should not be instantiated at all, serving only as a container for static members or constants. By using a private constructor, you ensure that the class cannot be instantiated accidentally.

constructor overloading - having more than one constructor with different parameters so that every constructor can perform a different task.

Destructor

A destructor is a special member function that works just opposite to a constructor; unlike constructors that are used for initializing an object, destructors destroy (or delete) the object. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime.
start with tilde(~) sign.
no parameters allowed
no return type

```
#include <iostream>

class MyClass {
public:
    // Constructor
    MyClass() {
        std::cout << "Constructor called." << std::endl;
    }

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called." << std::endl;
    }
};

int main() {
    {
        MyClass obj; // Object created within a block
    } // Object goes out of scope, destructor is automatically called

    std::cout << "After the block." << std::endl;

    return 0;
}

output:
Constructor called.
Destructor called.
After the block.
```

when object is created statically then the object get destroyed automatically(destructor is called automatically).
A destructor function is called automatically when:
➔ the object goes out of scope
➔ the program ends
➔ a scope (the { } parenthesis) containing local variable ends.
➔ a delete operator is called

when object is created dynamically then the we need to maually destroy the object.

```
delete object_name;
```

what is object?

An object is an instance of a Class. It is an identifiable entity with some characteristics and behavior. Objects are the basic units of object-oriented programming.

```
// Syntax to create an object in C++:
class_name objectName;
or
class_name objectName = class_name();
// Syntax to create an object dynamically in C++:
class_name* objectName = new class_name();


class A{
    public:
    int a;
    A(){

    }
    void f1(){

    }
};

int main(){
    // static creation of object
    A obj1;
    // static creation of object
    A obj2 = A();

    // dynamic creation of object
    A* obj3 = new A();

    return 0;
}
```

why do we need oops?

❖ To make the development and maintenance of projects more effortless.
❖ To provide the feature of data hiding that is good for security concerns.
❖ We can solve real-world problems if we are using object-oriented programming.
❖ It ensures code reusability.
❖ It lets us write generic code: which will work with a range of data, so we don't have to write basic stuff over and over again.
❖ Problems can be divided into subparts.
❖ It increases thereadability, understandability, andmaintainability of the code.
❖ Data and code are bound together by encapsulation.

disadvantages of opps:

❖ Requires pre-work and proper planning.
❖ In certain scenarios, programs can consume a large amount of memory.

❖ Not suitable for a small problem.
❖ Proper documentation is required for later use.

Diffenences between class and object.

class is a blueprint of object and used to create object while object is an instance of the class.
no memory is allocated when a class is created but memory is created when object is created.

Difference between class and structures.

By default all the members in class are private while it is public in case of stuctures.

padding

processor doesnot read 1 byte at a time from memory it reads 1 word at a time.
1 word = 4 bytes in 32-bit architechture.
1 word = 8 bytes in 64-bit architechture.
in one cycle one word is read by the processor from the memory.

Size of char : 1
Size of short int : 2
Size of int : 4
Size of long : 8
Size of float : 4
Size of double : 8

each data type start its memory allocation from its multiple.

```
#include <bits/stdc++.h>
using namespace std;

class Employee{
    char name;
    char x;
    short int age;
    short int val;
    int y;
};

int main(){
    Employee aman;
    cout<<sizeof(aman)<<endl;
    return 0;
}
```

it is done to save cpu cycle.

If want to save memory use #pragma pack(1) above class.

Access Modifiers

It is used to assign access to the class members. It sets some restrictions on the class members from accessing the outside functions directly.
Public: All the class members and methods can be accessed everywhere (inside and outside the class).
Private: All the class members and methods can be acceessed only inside the same class.
By default all the members and methods are private.
Protected: All the class members and methods can be acceessed in the same class and derived/child class.

```
class person {
    // nothing written so private
        int a;
    private:
        int b;  // private
    public:
        int c;  // public
    protected:
        int d;  // protected
};
```

this

this pointer holds the address of the current object. In simple words, you can say that this pointer points to the current object of the class. It allows the member function to access the data members and member functions of the object it is associated with.
There can be three main usages of this keyword in C++.
● It can be used to refer to a current class instance variable.
● It can be used to pass the current object as a parameter to another method.
● It can be used to declare indexers.

```
#include <bits/stdc++.h>
using namespace std;

class person{
    string name;
    int age;
    public:
    person(string name, int age){
        this->name = name;
        this->age = age;
        cout<<"Name: "<<name<<" Age: "<<age<<endl;
    }
};

int main(){
    person obj("Ram", 24);
    return 0;
}
```

Shallow Copy

An object is created by simply copying the data of all variables of the original object. Here, the pointer will be copied but not the memory it points to. It means that the original object and the created copy will now point to the same memory address, which is generally not preferred. Since both objects will reference the exact memory location, then change made by one will reflect those change in another object as well. This can lead to unpleasant side effects if the elements of values are changed via some other reference. Since we wanted to create an object replica, the Shallow copy will not fulfill this purpose. Note: C++ compiler implicitly creates a copy constructor and assignment operator to perform shallow copy at compile time. A shallow copy can be made by simply copying the reference.

```
class student {
    int age;
    char* name;
    public:
    student(int age, char* name){
        this->age = age;
        this->name = name;
    }
};
```

Deep Copy

An object is created by copying all the fields, and it also allocates similar memory resources with the same value to the object. To perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is necessary to allocate memory to the other

constructors' variables dynamically. A deep copy means creating a new array and copying over the values. Changes to the array values referred to will not result in changes to the array data refers to.

```
class student {
    int age;
    char* name;
    public:
    student(int age, char* name){
        this->age = age;
        this->name = new char[strlen(name) +1];
        strcopy(this->name, name);
    }
};
```

Const Keyword

This is used to indicate that a variable, function parameter, or member function does not modify the object's state or that the variable itself is immutable.

```
#include <bits/stdc++.h>
using namespace std;

class MyClass
{
    const int a = 10;
public:
    void regularFunction()
    {
        cout<<a<<endl;
        // Can modify data members of the class here
    }

    void constFunction() const
    {
        // Cannot modify data members of the class here
    }
};

int main()
{
    MyClass ob;
    // both const and not const function can be called.
    ob.constFunction();
    ob.regularFunction();
    const MyClass obj;
    // Error: Cannot call a non-const member function on a const object
    obj.regularFunction();
    // Okay: Can call a const member function on a const object
    obj.constFunction();
}
```

Initialisation List

It is a special syntax used in constructors to initialize the member variables of a class before the body of the constructor executes. It is an efficient and preferred way to initialize class members, especially for non-default constructors or when dealing with class members that are objects themselves.
It is placed after the constructor's argument list but before the constructor's body, and it is specified using a colon : followed by a comma-separated list of member variable initializations.
Methods cannot be intialised here.

```
#include <iostream>

class MyClass {
private:
    int x;
    double y;

public:
    // Constructor using initialization list
    MyClass(int a, double b) : x(a), y(b) {
        // Constructor body (optional)
    }

    void printValues() {
        std::cout << "x: " << x << ", y: " << y << std::endl;
    }
};

int main() {
    MyClass obj(42, 3.14);
    obj.printValues();

    return 0;
}
```

Scope resolution operator ::

It is used to access elements (such as variables, functions, or types) that are defined in a specific scope. It allows you to explicitly specify which scope or namespace a particular identifier belongs to, resolving any potential naming conflicts and providing access to elements defined in different scopes.

```
#include <bits/stdc++.h>
using namespace std;

class human{
    public:
    int age;
    human(){
        cout<<"human constructor called"<<endl;
    }
};

class male: public human{
    public:
    string col;
    male(){
        cout<<"male constructor called"<<endl;
    }
};

class D{
    public:
    void func(){
        cout<<"D called"<<endl;
    }
};

class A : public D{
    public:
    void func(){
        cout<<"A called"<<endl;
    }
};

class B{
    public:
    void func(){
        cout<<"B called"<<endl;
    }
};

class C: public A, public B{
    public:
    void func(){
        cout<<"C called"<<endl;
    }
};

int main(){
    C obj;
    obj.func();
```

```
    obj.A::func();
    obj.B::func();
    obj.A::D::func();
    return 0;
}
```

Static Keyword

It is used to define class-level or object-independent variables and member functions. When static is applied to variables, it means that the variable belongs to the class itself, not to any specific instance (object) of the class. When static is applied to member functions, it means the function can be called directly through the class name, without needing an object of the class.

Properties of static:

Static Variables: A static variable is shared among all instances (objects) of the class. There is only one instance of the static variable for the entire class, regardless of how many objects are created. A static variable is initialized only once at the start of the program and retains its value between function calls.

Static Member Functions: A static member function does not have a this pointer and can be called without creating an instance of the class. It operates on class-level data and cannot access non-static member variables or call non-static member functions.

```
#include <iostream>

class MyClass {
public:
    static int staticVar; // Static variable

    MyClass(int x) {
        staticVar = x;
    }

    static void staticFunction() {
        std::cout << "Static function, StaticVar: " << staticVar << std::endl;
    }

    void regularFunction() {
        std::cout << "Regular function, StaticVar: " << staticVar << std::endl;
    }
};

int MyClass::staticVar = 0;

int main() {
    MyClass::staticFunction(); // Output: Static function, StaticVar: 0

    MyClass obj(42);
    obj.regularFunction();     // Output: Regular function, StaticVar: 42

    MyClass::staticFunction(); // Output: Static function, StaticVar: 42

    return 0;
}
```

Singleton class

Singleton class is created only once, and the same instance is used throughout the program's execution. This follows the Singleton pattern, which ensures that there is only one instance of the class, making it useful for scenarios where you need a global point of access to a shared resource or functionality.

```
#include <iostream>

class Singleton {
private:
    static Singleton* instance; // Static member to hold the single instance

    // Private constructor to prevent direct instantiation
    Singleton() {
        std::cout << "Singleton instance created." << std::endl;
    }

public:
    // Static method to get the single instance
    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void showMessage() {
        std::cout << "Hello from Singleton!" << std::endl;
    }
};

// Initialize the static member variable to nullptr
Singleton* Singleton::instance = nullptr;

int main() {
    // Get the singleton instance using getInstance()
    Singleton* s1 = Singleton::getInstance();
    s1->showMessage(); // Output: Hello from Singleton!

    // Since it is a singleton, both pointers point to the same instance
    Singleton* s2 = Singleton::getInstance();
    s2->showMessage(); // Output: Hello from Singleton!

    // Both pointers have the same memory address
    std::cout << "Address of s1: " << s1 << std::endl;
    std::cout << "Address of s2: " << s2 << std::endl;

    return 0;
}
```

Encapsulation

Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.
fully encapsualted : all data members or variables are private.

```
class Student
{
    // private data members
private:
    string studentName;
    int studentRollno;
    int studentAge;
    // get method for student name to access
    // private variable studentName
public:
    string getStudentName(){
        return studentName;
    }
    // set method for student name to set
    // the value in private variable studentName
    void setStudentName(string studentName){
        this->studentName = studentName;
    }
    // get method for student rollno to access
    // private variable studentRollno
    int getStudentRollno(){
        return studentRollno;
    }
    // set method for student rollno to set
    // the value in private variable studentRollno
    void setStudentRollno(int studentRollno){
        this->studentRollno = studentRollno;
    }
    // get method for student age to access
    // private variable studentAge
    int getStudentAge(){
        return studentAge;
    }
    // set method for student age to set
    // the value in private variable studentAge
    void setStudentAge(int studentAge){
        this->studentAge = studentAge;
    }
};
```

Abstraction - implementation hiding.

Advantages Of Abstraction

- Only you can make changes to your data or function, and no one else can.
- It makes the application secure by not allowing anyone else to see the background details.
- Increases the reusability of the code.
- Avoids duplication of your code.

```
class abstraction{
    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;
    }
};
```

Inheritence

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class). The derived class inherits the features from the base class and can have additional features of its own. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.
parent class / base class / super class
child class / sub class

```
class parent_class{
    //Body of parent class
};

class child_class: access_modifier parent_class {
    //Body of child class
};
```

parent\child	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	NA	NA	NA

Types of Inheritence

1. single inheritence - In single inheritance, one class can extend the functionality of another class. There is only one parent class and one child class in single inheritances.
2. multilevel inheritence - When a class inherits from a derived class, and the derived class becomes the base class of the new class, it is called multilevel inheritance. In multilevel inheritance, there is

more than one level.

3. multiple inheritence - inherit from two or more super class. have more than one parent. In multiple inheritance, a class can inherit more than one class. This means that a single child class can have multiple parent classes in this type of inheritance.
4. hierarchical inheritence - parent have more than one child. In hierarchical inheritance, one class is a base class for more than one derived class.
5. Hybrid inheritence - combination of two or more inheritence. Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.

Inheritence ambiguity.

- can be solved using scope resolution operator.

polymorphism

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism is a concept that allows you to perform a single action in different ways. Polymorphism is the combination of two Greek words. The poly means many, and morphs means forms. So polymorphism means many forms.

1. **Compile time polymorphism / Static polymorphism.**
    1. function overloading - When there are multiple functions in a class with the same name but different parameters, these functions are overloaded. The main advantage of function overloading is that it increases the program's readability. Functions can be overloaded by using different numbers of arguments or by using different types of arguments.
    2. operator overloading.
2. **Run time polymorphism / Dynamic polymorphism.**
    1. Method overriding - Method overriding is a feature that allows you to redefine the parent class method in the child class based on its requirement. In other words, whatever methods the parent class has by default are available in the child class. But, sometimes, a child class may not be satisfied with parent class method implementation. The child class is allowed to redefine that method based on its requirement. This process is called method overriding.
    Rules for method overriding:
    ● The parent class method and the method of the child class must have the same name.
    ● The parent class method and the method of the child class must have the same parameters.
    ● It is possible through inheritance only.

```
class Parent{
    public:
    void show() {
        cout<<"Inside parent class"<<endl;
    }
};
class subclass1: public Parent {
    public:
    void show() {
        cout<<"Inside subclass1"<<endl;
    }
};
class subclass2: public Parent {
    public:
    void show() { cout<<"Inside subclass2";
    }
};
```

Differences between compile time and runtime polymorphism.

Differences between abstraction and encapsulation.

Virtual Function

A virtual function is a member function in the base class that we expect to redefine in derived classes. It is declared using the virtual keyword. A virtual function is used in the base class to ensure that the function is overridden. This especially applies to cases where a pointer of base class points to a derived class object.
C++ determines which function is invoked at the runtime based on the type of object pointed by the base class pointer when the function is made virtual.

```cpp
#include <iostream>

class Shape {
public:
    // Virtual function for calculating the area (can be overridden in derived classes)
    virtual double getArea() {
        return 0.0; // Base class provides a default implementation
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    // Override the getArea() function to provide Circle-specific implementation
    double getArea() {
        return 3.14 * radius * radius;
    }
};

class Rectangle : public Shape {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Override the getArea() function to provide Rectangle-specific implementation
    double getArea() {
        return length * width;
    }
};

int main() {
    Shape* shape1 = new Circle(5.0);
    Shape* shape2 = new Rectangle(4.0, 3.0);
    Shape shape3 = Rectangle(5, 3);

    std::cout<<shape3.getArea()<<std::endl;

    // Polymorphic behavior: getArea() depends on the actual object type
    std::cout << "Area of Circle: " << shape1->getArea() << std::endl;       // Output: Area of Circle: 78.5
    std::cout << "Area of Rectangle: " << shape2->getArea() << std::endl;    // Output: Area of Rectangle: 12

    delete shape1;
    delete shape2;
```

```
    return 0;
}
```

Diamond Problem

It is a classic issue that arises in some object-oriented programming languages, including C++, when multiple inheritance is used. It occurs when a class inherits from two or more base classes that have a common ancestor. This results in ambiguity when accessing the members of the common ancestor through the derived class.

What is a pure virtual function?

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have an implementation. We only declare it. A pure virtual function is declared by assigning 0 in the declaration. A pure virtual function (or abstract function) in C++ is a virtual function for which we can implement, But we must override that function in the derived class; otherwise, the derived class will also become an abstract class.

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

Abstract Class

Abstract classes can't be instantiated, i.e., we cannot create an object of this class. However, we can derive a class from it and instantiate the object of the derived class. An Abstract class has at least one pure virtual function.

Properties of the abstract classes:

❖ It can have normal functions and variables along with pure virtual functions.
❖ Prominently used for upcasting(converting a derived-class reference or pointer to a base-class. In other words, upcasting allows us to treat a derived type as a base type), so its derived classes can use its interface.
❖ If an abstract class has a derived class, they must implement all pure virtual functions, or they will become abstract.

```
#include<iostream>
using namespace std;
class Base{
    public:
    virtual void s() =0;// Pure Virtual Function
};
class Derived: public Base {
    public:
    void s() {
        cout<<"Virtual Function in Derived_class";
        }
};

int main() {
    Base *b;
    Derived d_obj;
    b = &d_obj;
    b->s();
}
```

If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class. We cannot create objects of an abstract class.
However, we can derive classes from them and use their data members and member functions (except pure virtual functions).

Friend Funtion

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function. A class's friend function is defined outside that class's scope, but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions. A friend function in C++ is a function that is preceded by the keyword "friend."

```
#include <bits/stdc++.h>
using namespace std;

class Rectangle{
    private:
    int length;
    public:
    Rectangle(){
        length = 10;
    }
    friend int printLength(Rectangle); // friend function
};

int printLength(Rectangle obj){
    obj.length += 10;
    return obj.length;
}

int main(){
    Rectangle obj;
    cout<<"Length of Rectangle: "<<printLength(obj)<<endl;
    return 0;
}
```

Characteristics of friend function:

● A friend function can be declared in the private or public section of the class.
● It can be called a normal function without using the object.
● A friend function is not in the scope of the class, of which it is a friend.
● A friend function is not invoked using the class object as it is not in the class's scope.
● A friend function cannot access the private and protected data members of the class directly. It needs to make use of a class object and then access the members using the dot operator.
● A friend function can be a global function or a member of another class.