Definition of oops.

* OOPs refers to the language that uses the concept of class and object in programming.
* The main objective of OOPs is to implement real-world entities such as polymorphism, inheritance, encapsulation, abstraction, etc.
* The main aim of 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 that function.

class and objects

* A Class is a user-defined data type that has data members and member functions.
* Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define the properties and behavior of the objects in a Class.
* An Object is an identifiable entity with some characteristics and behaviour. 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.

Why do we need object-oriented programming?

❖To make the development and maintenance of projects more effortless.

❖Data and code are bound together by encapsulation.

❖Code can be reused, and it reduces redundancy.

❖It also helps to hide unnecessary details with the help of Data Abstraction.

❖Problems can be divided into subparts.

❖It increases the readability, understandability, and maintainability of the code.

Data encapsulation is supported with “class”. The class consists of both data and functions. The data in a class is called a member, while functions in the class are called methods.

**Data Members:-**The variables which are declared in any class by using any fundamental data types (like int, char, float, etc.) or derived data types (like class, structure, pointer, etc.) are known as Data Members.

**Methods:-**A method is the equivalent of a function in object-oriented programming that is used inside classes. The methods are the actions that perform operations. A method accepts parameters as arguments, manipulates these, and then produces an output when the method is called on an object.

**Constructor :**- Constructors are special class functions that 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.

Example of smartphone

#include <iostream>

using namespace std;

class Smartphone {

    // Data Members (Properties)

    string model;

    int year\_of\_manufacture;

    bool \_5g\_supported;

public:

    // Constructor

    Smartphone(string mod, int manu, bool \_5g\_supp) {

        // Initializing data members

        model = mod;

        year\_of\_manufacture = manu;

        \_5g\_supported = \_5g\_supp;

    }

    // Methods

    void print\_details() {

        cout << "Model: " << model << endl;

        cout << "Year of Manufacture: " << year\_of\_manufacture << endl;

        cout << "5G Supported: " << (\_5g\_supported ? "Yes" : "No") << endl;

    }

};

int main() {

    // Example usage of the Smartphone class

    Smartphone myPhone("iPhone 13", 2022, true);

    myPhone.print\_details();

    return 0;

}

Another example

#include <iostream>

using namespace std;

// Creating class

class Smartphone {

    // Data Members (Properties)

    string model;

    int year\_of\_manufacture;

    bool \_5g\_supported;

public:

    // Constructor

    Smartphone(string model\_string, int manufacture, bool \_5g\_) {

        // Initializing data members

        model = model\_string;

        year\_of\_manufacture = manufacture;

        \_5g\_supported = \_5g\_;

    }

    // Methods

    void print\_details() {

        cout << "Model: " << model << endl;

        cout << "Year of Manufacture: " << year\_of\_manufacture << endl;

        cout << "5G Supported: " << (\_5g\_supported ? "Yes" : "No") << endl;

    }

};

int main() {

    // Creating objects of Smartphone class

    Smartphone iphone("iPhone 11", 2019, false);

    Smartphone redmi("Redmi Note 11T", 2021, true);

    Smartphone oneplus("OnePlus Nord", 2020, true);

    // Accessing class variables

    int iphone\_manufacture\_date = iphone.year\_of\_manufacture;

    bool redmi\_support\_5g = redmi.\_5g\_supported;

    string oneplus\_model = oneplus.model;

    // Calling methods on objects

    iphone.print\_details();

    redmi.print\_details();

    oneplus.print\_details();

    return 0;

}

Access Modifiers

In C++, access modifiers allow us to control the visibility and accessibility of class members (variables and functions) within and outside the class. There are three primary access modifiers available:

1. **\*\*Public:\*\***

- Members with the `public` modifier can be accessed from anywhere, both inside and outside the class.

- Example

class Person {

public:

string name;

};

2. **\*\*Private:\*\***

- Members with the `private` modifier can only be accessed by member functions inside the class. They are not accessible from outside the class.

- Example:

```cpp

class Person {

private:

int fb\_password;

}

3. **\*\*Protected:\*\***

- Members with the `protected` modifier are accessible within the class and can also be accessed from derived classes (child classes or subclasses). If you don't create a derived class, they cannot be accessed from outside the class.

- Example:

```cpp

class Person {

protected:

string assets;

};

By default, in C++, all class members are private if you don't explicitly specify an access specifier.

Examples

#include <iostream>

using namespace std;

class Smartphone {

    string model;             // Private by default

public:

    int year\_of\_manufacture;  // Public data member

protected:

    string company\_name;      // Protected data member

private:

    int password;             // Private data member

public:

    // Constructor

    Smartphone(string mod, int manu, string company, int pass) {

        model = mod;

        year\_of\_manufacture = manu;

        company\_name = company;

        password = pass;

    }

    // Public method

    void call() {

        cout << "Calling from " << company\_name << " " << model << endl;

    }

    // Protected method

    void about\_phone() {

        cout << "About " << company\_name << " " << model << endl;

    }

private:

    // Private method

    void unlock\_lockscreen() {

        cout << "Unlocking " << company\_name << " " << model << " lock screen" << endl;

    }

};

int main() {

    // Create objects of the Smartphone class

    Smartphone iPhone("iPhone 13", 2022, "Apple", 1234);

    Smartphone Samsung("Galaxy S21", 2021, "Samsung", 5678);

    // Access public data members and call public methods

    cout << "Year of Manufacture: " << iPhone.year\_of\_manufacture << endl;

    iPhone.call();

    cout << "Year of Manufacture: " << Samsung.year\_of\_manufacture << endl;

    Samsung.call();

    // Attempting to access protected and private members/methods will result in a compilation error

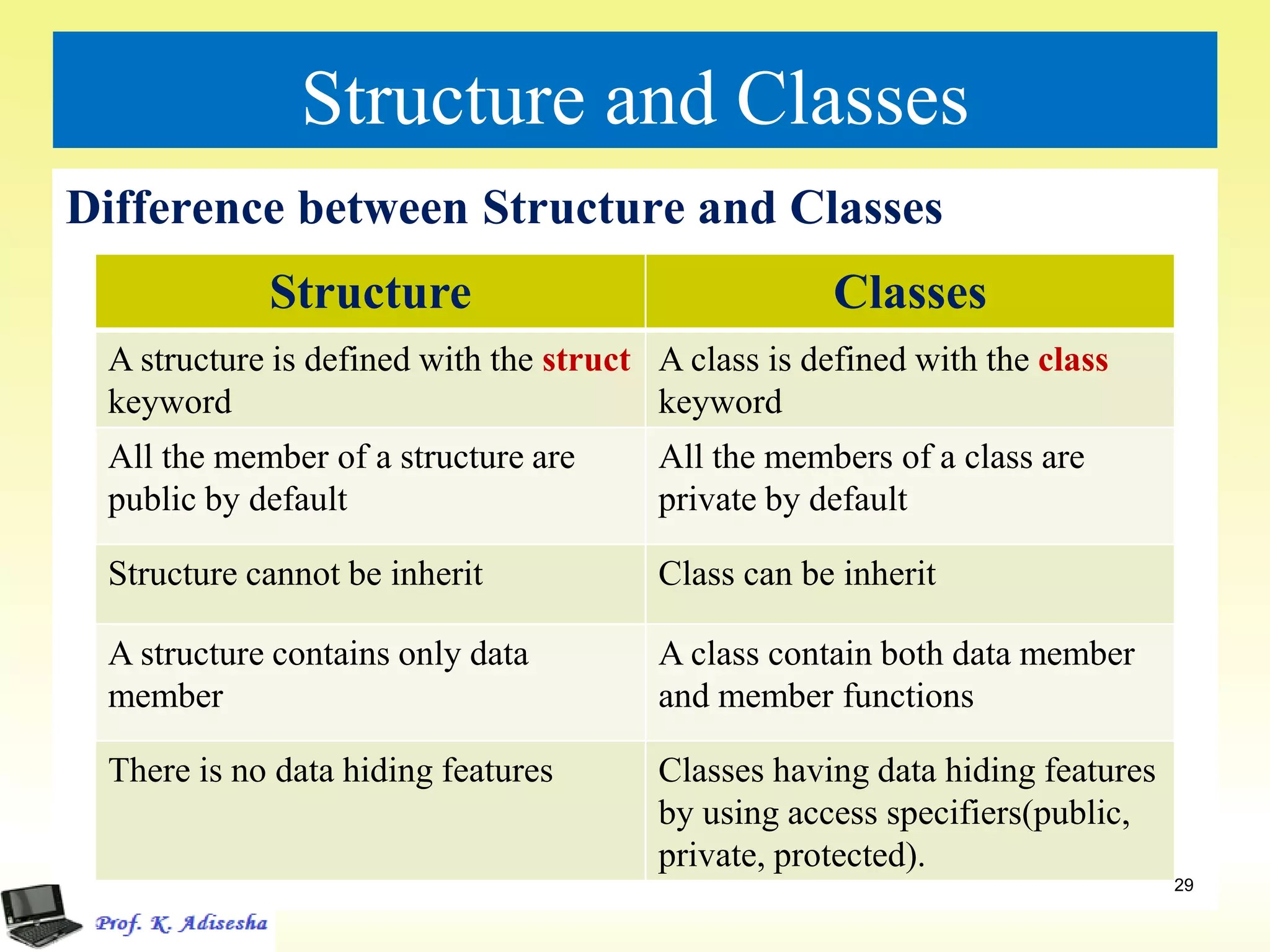
    // iPhone.about\_phone(); // This line will cause a compilation error

    // iPhone.unlock\_lockscreen(); // This line will cause a compilation error

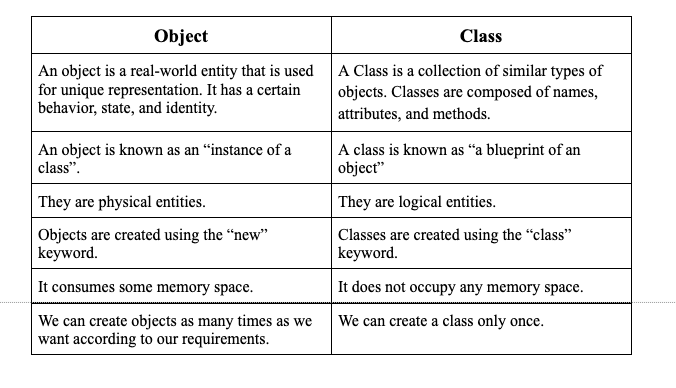
    return 0;

}

**Difference between class and structure**



**Difference between object and class**



Constructors

A constructor is a special member function automatically called when an object is created. it does not have any return type. It has the same name as the class itself.

A constructor initializes the class data members with garbage value if we don’t put any value to it explicitly.

If we do not specify a constructor, the C++ compiler generates a default constructor for an object (which expects no parameters and has an empty body).

Types of Constructors: There are three types of constructors in C++:

★Default constructor

★Parameterized Constructor

★Copy Constructor

**Default constructor:**

A constructor that doesn't take any argument or has no parameters is known as a default constructor.

public:

    // Default constructor

    class\_name() {

        // Initializing data members with their default values

        data\_member1 = 69;

        data\_member2 = "Coding Ninjas";

    }

};

Here, the class\_name() constructor will be called when the object is created. This sets the data\_member1 variable of the object to 69 and the data\_member2 variable of the object to “Coding Ninjas”.

Note: If we have not defined a constructor in our class, the C++ compiler will automatically create a default constructor with an empty code and no parameters, which will initialize data members with garbage values. When we write our constructor explicitly, the inbuilt constructor will not be available for us.

**Parameterized Constructor:**

This is another type of Constructor with parameters. The parameterized constructor takes its arguments provided by the programmer.

**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.

class class\_name {

private:

    int data\_member1;

    std::string data\_member2;

public:

    // Copy constructor

    class\_name(class\_name& obj) {

        // Copies data of the obj parameter

        data\_member1 = obj.data\_member1;

        data\_member2 = obj.data\_member2;

    }

};

Syntax , ex and output all three constructors

#include <iostream>

#include <string>

class smartphone {

private:

    std::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(std::string model\_string, int manufacture, bool \_5g\_) {

        model = model\_string;

        year\_of\_manufacture = manufacture;

        \_5g\_supported = \_5g\_;

    }

    // Copy constructor

    smartphone(const smartphone& obj) {

        model = obj.model;

        year\_of\_manufacture = obj.year\_of\_manufacture;

        \_5g\_supported = obj.\_5g\_supported;

    }

    // Display smartphone information

    void displayInfo() {

        std::cout << "Model: " << model << std::endl;

        std::cout << "Year of Manufacture: " << year\_of\_manufacture << std::endl;

        std::cout << "5G Supported: " << (\_5g\_supported ? "Yes" : "No") << std::endl;

    }

};

int main() {

    // Using default constructor

    std::cout << "Default Constructor:" << std::endl;

    smartphone default\_smartphone;

    default\_smartphone.displayInfo();

    // Using parameterized constructor

    std::cout << "\nParameterized Constructor:" << std::endl;

    smartphone parameterized\_smartphone("iPhone 12", 2020, true);

    parameterized\_smartphone.displayInfo();

    // Using copy constructor

    std::cout << "\nCopy Constructor:" << std::endl;

    smartphone copy\_smartphone(parameterized\_smartphone);

    copy\_smartphone.displayInfo();

    return 0;

}

Default Constructor:

Model: unknown

Year of Manufacture: 0

5G Supported: No

Parameterized Constructor:

Model: iPhone 12

Year of Manufacture: 2020

5G Supported: Yes

Copy Constructor:

Model: iPhone 12

Year of Manufacture: 2020

5G Supported: Yes

Constructor Overloading:

Constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

#include <iostream>

#include <string>

class smartphone {

private:

    std::string model;

    int year\_of\_manufacture;

    bool \_5g\_supported;

public:

    // Constructor with 0 parameters

    smartphone() {

        model = "unknown";

        year\_of\_manufacture = 0;

        \_5g\_supported = false;

    }

    // Constructor with 2 parameters

    smartphone(std::string model\_string, bool \_5g\_) {

        model = model\_string;

        \_5g\_supported = \_5g\_;

    }

    // Constructor with 3 parameters

    smartphone(std::string model\_string, int manufacture, bool \_5g\_) {

        model = model\_string;

        year\_of\_manufacture = manufacture;

        \_5g\_supported = \_5g\_;

    }

};

int main() {

    // Creating objects of smartphone class

    // Using constructor with 0 parameters

    smartphone unknown;

    // Using constructor with 2 parameters

    smartphone redmi("Note 7 Pro", false);

    // Using constructor with 3 parameters

    smartphone iphone("iPhone 11", 2019, false);

    return 0;

}

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.

When is a destructor called?

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

**Rules**

❖A destructor should be declared in the public section of the class.

❖The programmer cannot access the address of the destructor.

❖It has no return type, not even void.

❖When you do not specify any destructor in a class, the compiler generates a default destructor and inserts it into your code.

#include <iostream>

class Guided\_path {

public:

    // Constructor

    Guided\_path() {

        std::cout << "Constructor is called" << std::endl;

        std::cout << "Welcome to Guided Path" << std::endl;

    }

    // Destructor

    ~Guided\_path() {

        std::cout << "Happy Learning" << std::endl;

        std::cout << "Destructor is called" << std::endl;

    }

};

int main() {

    // Object created

    Guided\_path obj;

    // At the end, the object is destructed

    return 0;

}

Constructor is called

Welcome to Guided Path

Happy Learning

Destructor is called

When should the destructor use delete to free the memory?

If the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

This Pointer:

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.

Class mobile{

String model;

int year\_of\_manufacture;

public:

void set\_details(string model,int year\_of\_manufacture){

this->model = model;

this->year\_of\_manufacture = year\_of\_manufacture;

}

Void print(){

Cout <<this->model << endl;

Cout <<this->year\_of\_manufacture <<endl;

}

};

This pointer is available only within the non-static member functions of a class. If the member function is static , it will be common to all the objects, and hence a single object cant’t refer to those functins independently

Shallow and Deep Copy

**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.

**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.

Code eg: <https://www.codingninjas.com/studio/guided-paths/basics-of-c/content/118817/offering/1382186?leftPanelTab=0>

**When is it necessary to use this pointer?**

Suppose we have two local variables with the same name as the data members’ names. Suppose you want to assign the local variable value to the data members. In that case, you won’t be able to do until unless you use this pointer because the compiler won’t know that you are referring to the object’s data members unless you use this pointer.

Encapsulation:

Encapsulation is about wrapping data and methods into a single class and protecting it from outside intervention.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. Now,

The finance section handles all the financial transactions and keeps records of all the data related to finance.

Similarly, the sales section handles all the sales-related activities and keeps records of all the sales.

Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data.

**Advantages:**

Encapsulation provides several benefits, including:

* Improved code maintainability: Encapsulation helps in improving the code maintainability by providing a clear separation between the implementation details of a class and its clients.
* Data hiding: Encapsulation enables data hiding, which protects the data members of a class from being accessed and modified by the clients of the class.
* Code reuse: Encapsulation helps in code reuse by providing a modular design that can be easily extended and modified.
* Security: Encapsulation provides security by preventing unauthorized access to the data members of a class.
* #include <iostream>
* using namespace std;
* class Student {
* private:
* string studentName;
* int studentRollno;
* int studentAge;
* public:
* string getStudentName() {
* return studentName;
* }
* void setStudentName(string studentName) {
* this->studentName = studentName;
* }
* int getStudentRollno() {
* return studentRollno;
* }
* void setStudentRollno(int studentRollno) {
* this->studentRollno = studentRollno;
* }
* int getStudentAge() {
* return studentAge;
* }
* void setStudentAge(int studentAge) {
* this->studentAge = studentAge;
* }
* };
* int main() {
* Student obj;
* obj.setStudentName("Avinash");
* obj.setStudentRollno(101);
* obj.setStudentAge(22);
* cout << "Student Name : " << obj.getStudentName() << endl;
* cout << "Student Rollno : " << obj.getStudentRollno() << endl;
* cout << "Student Age : " << obj.getStudentAge();
* return 0;
* }

Abstraction:

Abstraction is selecting data from a larger pool to show only relevant details of the object to the user. It helps in reducing programming complexity and efforts. It is one of the most important concepts of OOPs.

Real-life example: When you send an email to someone, you just click send, and you get the success message; what happens when you click send, how data is transmitted over the network to the recipient is hidden from you (because it is irrelevant to you).

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data members will be visible to the outside world and not. Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members.

**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.

Inheritance

Inheritance is the process of inheriting the properties and behaviour of an existing class into a new/derived class. The derived class inherits the features from the base class and can have additional features of its own.

Syntax:

class parent\_class{

    //Body of parent class

};

class child\_class: access\_modifier parent\_class {

    //Body of child class

};

**Modes of Inheritance**

**1. Public mode:**

If we derive a subclass from a public base class. Then, the base class’s public members will become public in the derived class, and protected class members will become protected in the derived class.

**2. Protected mode**:

If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

**3. Private mode:**

If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

class Vehicle {

public:

    string color;

    int max\_speed;

};

class Car : public Vehicle {

public:

    int num\_gears;

};

class Bicycle : public Vehicle {

public:

    bool is\_foldable;

};

class Truck : public Vehicle {

public:

    int max\_weight;

};

int main() {

    Car myCar;

    myCar.color = "Red";

    myCar.max\_speed = 200;

    myCar.num\_gears = 5;

    Bicycle myBicycle;

    myBicycle.color = "Blue";

    myBicycle.max\_speed = 30;

    myBicycle.is\_foldable = true;

    Truck myTruck;

    myTruck.color = "Green";

    myTruck.max\_speed = 100;

    myTruck.max\_weight = 5000;

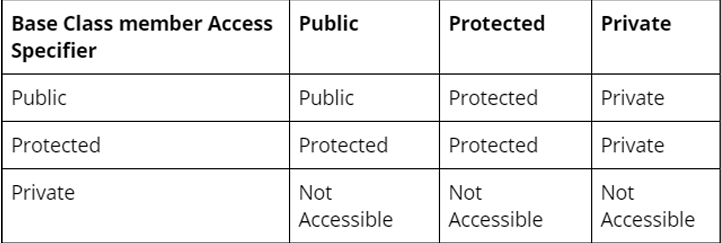
}

From above, we can see that two of the properties: Colour and MaxSpeed, are the same for every object. Hence, we can combine all these in one parent class and make three classes their subclass. This property is called Inheritance.

**Why we use inheritance:**

● The main advantage of inheritance is code reusability. We can reuse the code when we inherit the existing class’s methods and fields into a new class.

● The runtime polymorphism (method overriding) can be achieved by inheritance only.

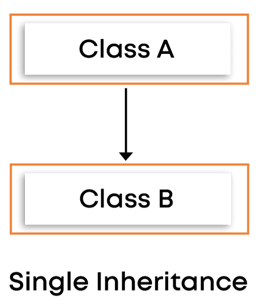


Types of Inheritance

C++ supports five types of inheritance they are as follows:

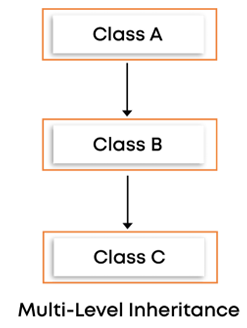
**1) Single inheritance**

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.



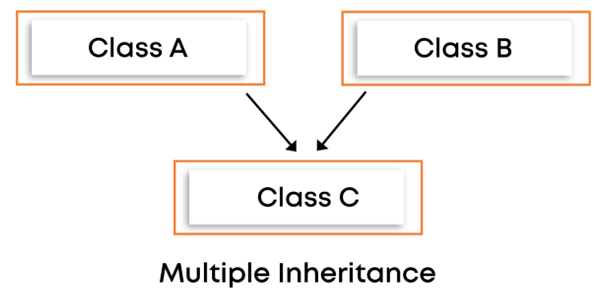
**2)Multilevel inheritance**

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.



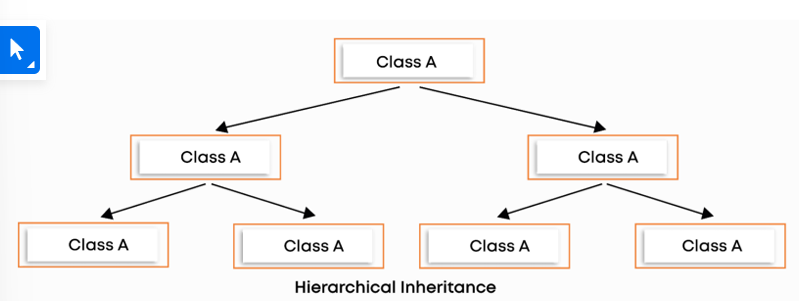
**3)Multiple inheritance**

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.



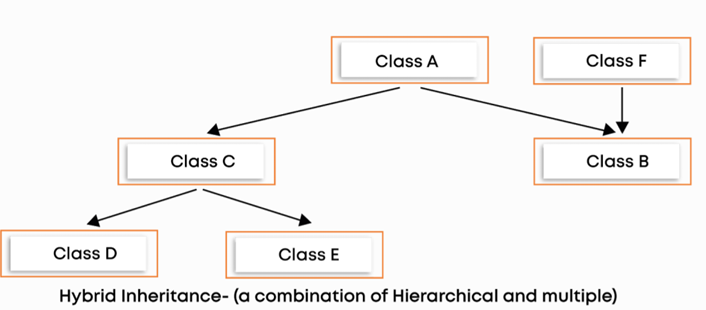
**4)Hierarchical inheritance**

In hierarchical inheritance, one class is a base class for more than one derived class.



**5)Hybrid inheritance**

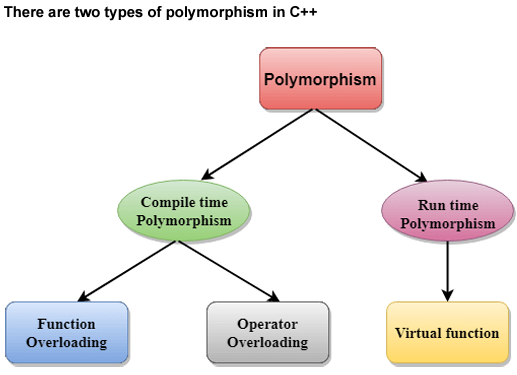
Hybrid inheritance is a combination of more than one type of inheritance. For example, A child and parent class relationship that follows multiple and hierarchical inheritances can be called hybrid inheritance.



Polymorphism

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.

Let’s understand polymorphism with a real-life example. Real-life example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, and an employee. So the same person possesses different behaviour in different situations. This is called polymorphism.



Compile time polymorphism

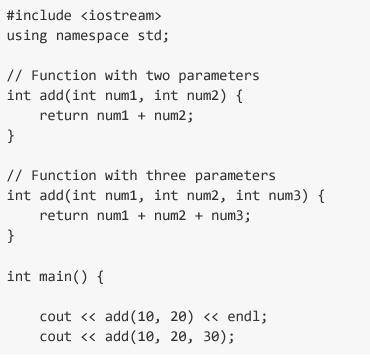
It is also known as static polymorphism. This type of polymorphism can be achieved through function overloading or operator overloading.

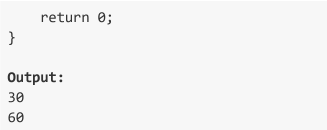
**a) Function overloading:**

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 it increases the readability of the program. Functions can be overloaded by using different numbers of arguments and by using different types of arguments.

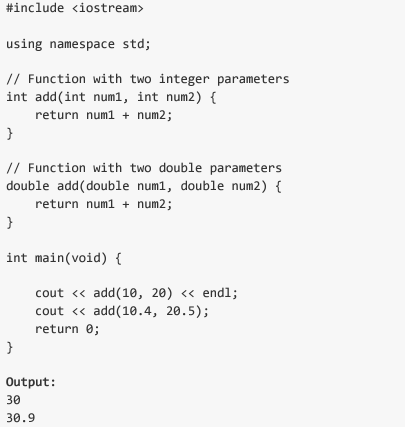
i) Function overloading with different numbers of arguments:

In this example, we have created two functions, the first add() performs the addition of the two numbers, and the second add() performs the addition of the three numbers. Let’s look at the example:





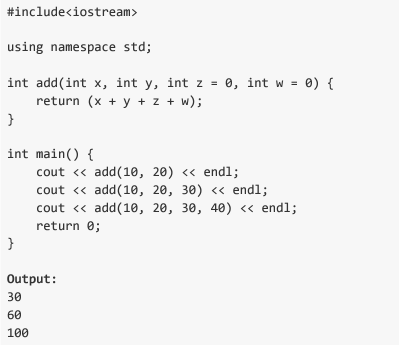
ii) Function overloading with different types of arguments: In this example, we have created two add() functions with different data types. The first add() takes two integer arguments and the second add() takes two double arguments.



Default Arguments:

A default argument is a value provided in a function declaration automatically assigned by the compiler if the function’s caller doesn’t provide a value for 2the argument with a default value. However, if arguments are passed while calling the function, the default arguments are ignored.

Example: A function with default arguments can be called with 2 or 3 or 4 arguments.



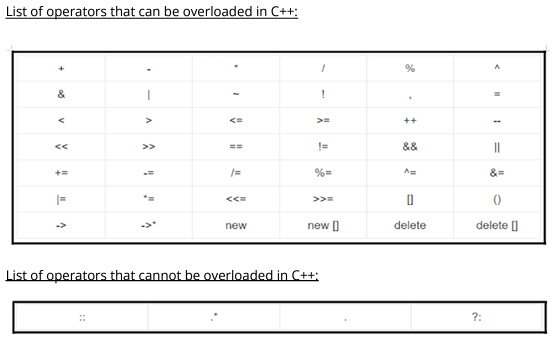
**b) Operator Overloading:**

C++ also provides options to overload operators. For example, we can make the operator (‘+’) for the string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. A single operator, ‘+,’ when placed between integer operands, adds them and concatenates them when placed between string operands. Points to remember while overloading an operator:

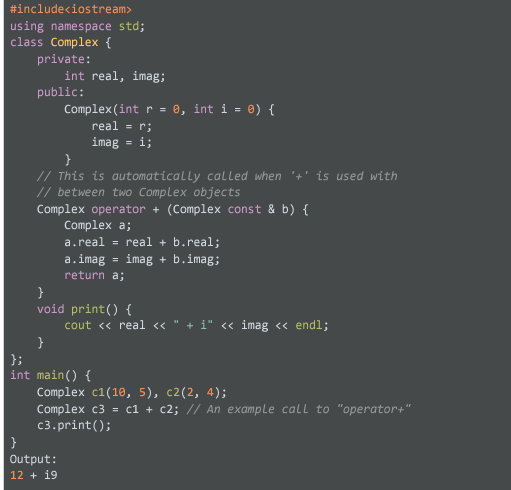
● It can be used only for user-defined operators (objects, structures) but cannot be used for in-built operators (int, char, float, etc.).

●Operators = and & are already overloaded in C++, so we can avoid overloading them.

● Precedence and associativity of operators remain intact.’



An eg to show the addition of two complex number



Run time Polymorphism

Runtime polymorphism is also known as dynamic polymorphism. Method overriding is a way to implement runtime polymorphism.

**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.

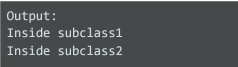
Rules for method overriding:

● 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.





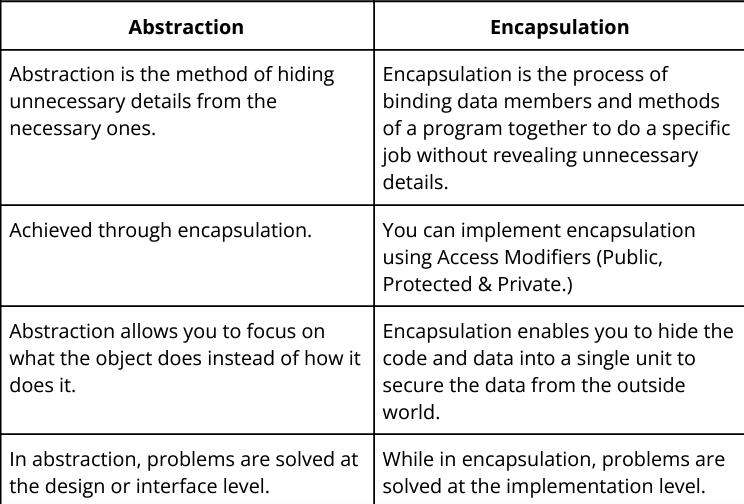
**Some Questions:**

**1.What is Encapsulation in C++? Why is it called Data hiding?**

The process of binding data and corresponding methods (behavior) into a single unit is called encapsulation in C++. we can keep variables and methods safes from outside interference and misuse.

If a field is declared private in the class, it cannot be accessed by anyone outside the class and hides the fields. Therefore, Encapsulation is also called data hiding.

**2. Difference between abstraction and encapsulation**



## **Advantages of Polymorphism**

Polymorphism offers the following advantages −

* It helps the programmer to reuse the codes, i.e., classes once written, tested and implemented can be reused as required. Saves a lot of time.
* Single variable can be used to store multiple data types.
* Easy to debug the codes.

Addition and multiplication of two complex numbers

#include <bits/stdc++.h>

#include<iostream>

using namespace std;

class ComplexNumbers {

    int real, imag;

    public:

    ComplexNumbers(int R, int I){

        real = R;

        imag = I;

    }

    void plus(ComplexNumbers c){

        this->real = this->real + c.real;

        this->imag = this->imag + c.imag;

    }

    void multiply(ComplexNumbers c){

        int temp1 = real\*c.real - imag\*c.imag;

        int temp2 = imag\*c.real + real\*c.imag;

        real = temp1;

        imag = temp2;

    }

    void print(){

        cout<<real<<" + i"<<imag;

    }

};

int main() {

    int real1, imaginary1, real2, imaginary2;

    cin >> real1 >> imaginary1;

    cin >> real2 >> imaginary2;

    ComplexNumbers c1(real1, imaginary1);

    ComplexNumbers c2(real2, imaginary2);

    int choice;

    cin >> choice;

    if (choice == 1) {

        c1.plus(c2);

        c1.print();

    } else if (choice == 2) {

        c1.multiply(c2);

        c1.print();

    } else {

        return 0;

    }

}

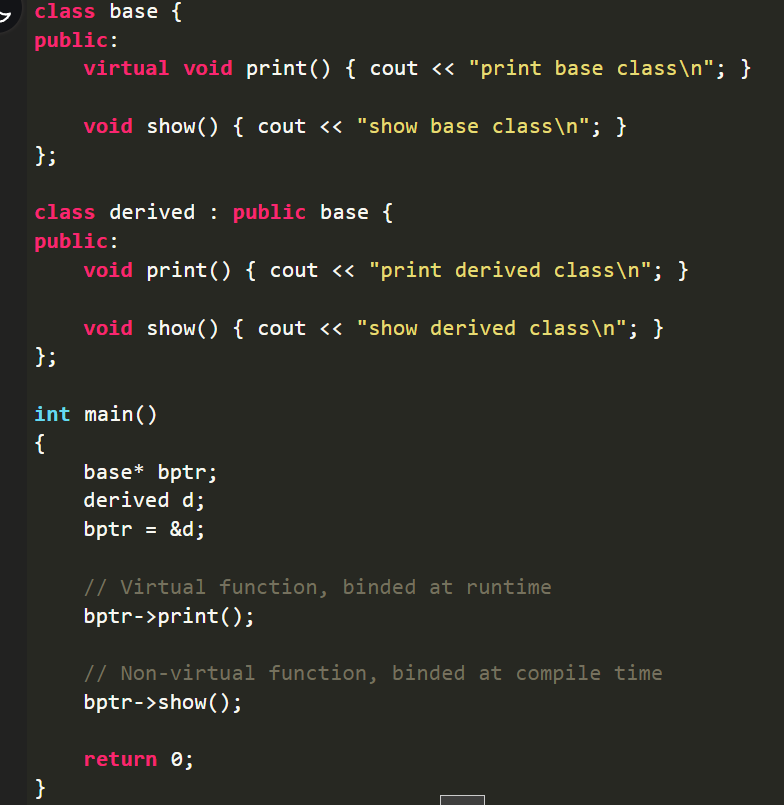
Virtual Function:

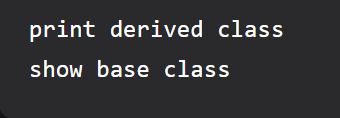
A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the method.

**Rules for Virtual Functions**

The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a [virtual destructor](https://www.geeksforgeeks.org/virtual-destructor/) but it cannot have a virtual constructor.





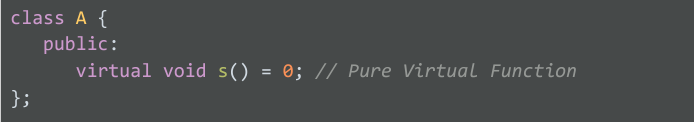
**Why use virtual functions**

We use virtual functions to ensure that the correct function is called for an object, regardless of the reference type used to call the function. They are basically used to achieve the runtime polymorphism and are declared in the base class by using the **virtual keyword** before the function.

Pure Virtual Function:

A pure virtual function is a virtual function in C++ for which we need not write any function definition and only have to declare it. It 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.



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.



**We can’t make instance of abstract base class**

Friend function: (need to understand more…..)

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 can be:

1. **A global function**
2. **A member function of another class**

1.

// C++ program to create a global function as a friend

// function of some class

#include <iostream>

using namespace std;

class base {

private:

    int private\_variable;

protected:

    int protected\_variable;

public:

    base()

    {

        private\_variable = 10;

        protected\_variable = 99;

    }

    // friend function declaration

    friend void friendFunction(base& obj);

};

// friend function definition

void friendFunction(base& obj)

{

    cout << "Private Variable: " << obj.private\_variable

        << endl;

    cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

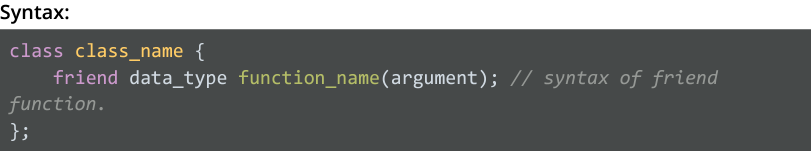
{

    base object1;

    friendFunction(object1);

    return 0;

}



**2. Member Function of Another Class as Friend Function**

We can also declare a member function of another class as a friend function in C++. The following example demonstrates how to use a member function of another class as a friend function in C++:

// C++ program to create a member function of another class

// as a friend function

#include <iostream>

using namespace std;

class base; // forward definition needed

// another class in which function is declared

class anotherClass {

public:

    void memberFunction(base& obj);

};

// base class for which friend is declared

class base {

private:

    int private\_variable;

protected:

    int protected\_variable;

public:

    base()

    {

        private\_variable = 10;

        protected\_variable = 99;

    }

    // friend function declaration

    friend void anotherClass::memberFunction(base&);

};

// friend function definition

void anotherClass::memberFunction(base& obj)

{

    cout << "Private Variable: " << obj.private\_variable

        << endl;

    cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

{

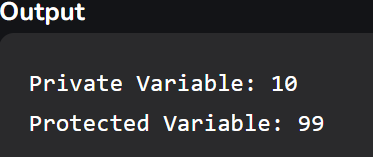
    base object1;

    anotherClass object2;

    object2.memberFunction(object1);

    return 0;

}



What are the characteristics of Friend Function?

★A friend function is not in the scope of the class, in which it has been declared as friend.

★It cannot be called using the object of that class.

★It can be invoked like a normal function without any object.

★Unlike member functions, it cannot use the member names directly.

★It can be declared in public or private parts without affecting its meaning. ★Usually, it has objects as arguments.