**Object-Oriented Programming (OOP)** is a programming paradigm or methodology that organizes and structures code using the concept of "**objects**". It is a widely used approach in software development because it helps manage complex systems and promotes code reusability, maintainability, and modularity.

#. Key principles and concepts:

| | |
|---|---|
| **Objects**: Objects are the fundamental building blocks of OOP. An object is a self-contained unit that combines data (attributes or properties) and behaviors (methods or functions) that operate on that data. Objects represent real-world entities or abstract concepts within a program. | **Classes**: A class is a user-defined data type. It consists of data members and member functions, which can be accessed and used by creating an instance of that class. It represents the set of properties or methods that are common to all objects of one type. A class is like a blueprint for an object. |
| **Inheritance**: Inheritance is a mechanism that allows one class (the subclass or derived class) to inherit the properties and methods of another class (the superclass or base class). This promotes code reuse and the creation of specialized classes that build upon existing functionality. | **Encapsulation**: Encapsulation is the process of combining data and functions into a single unit called **class**. In Encapsulation, the data is not accessed **directly**; it is accessed **through** the **functions** present inside the class. In simpler words, **attributes** of the class are kept **private** and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible. |
| **Abstraction**: Data abstraction refers to **providing** only **essential information** about the data to the outside world, **hiding** the **background** details or **implementation**. Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. | **Polymorphism**: The word polymorphism means having **many forms**. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. For example, A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person poses different behavior in different situations. This is called polymorphism. |

| | |
|---|---|
| **Class** | Think of a "human" as a class blueprint. It defines what **attributes** (characteristics) and **behaviors** (actions) a human can have. For example, a human class may have attributes like "name," "age," and "gender," and behaviors like "walking," "eating," and "talking." |
| | Think of "Nayan" as an instance of the "Person" class. The "Person" class defines **attributes** like "name" and "age" and **behaviors** like "speaking," "walking," and "eating." |

| | |
|---|---|
| **Object** | An "individual person" is an instance or object of the "human" class. Each person (object) has its own unique attributes (e.g., John, 30 years old) and can perform actions or behaviors (e.g., Mary is running). |
| | "Nayan" is an instance or object of the "Person" class. You, as an individual, have your own unique attributes (e.g., name: Nayan, age: 30) and can perform various actions (e.g., speaking, walking). |
| **Encapsulation** | Encapsulation is like keeping **personal information private**. In our "human" class, attributes like "name" and "age" are usually private, and we provide **public methods** (getters and setters) to access or **modify these attributes**, ensuring data integrity and control over access. |
| | Encapsulation is like keeping **personal details private**. Your personal information, such as your exact age and home address, is typically private. Public methods (getters and setters) are used to access or modify this information securely. |
| **Inheritance** | Inheritance can be seen as a **family tree**. Humans share common characteristics with their parents. For example, a "child" human inherits attributes and behaviors from their "parent" human. Inheritance helps create a **hierarchy of classes**, with shared and specialized features. |
| | Inheritance can be likened to family relationships. "Nayan" inherits attributes like "name" and "age" from your parents. There's a hierarchy, where you share characteristics with your ancestors, and you can also extend or specialize these attributes and behaviors. |
| **Polymorphism** | Polymorphism is the ability to **take on multiple forms**. Think of "talking" as a polymorphic behavior in the "human" class. Different people can talk in various ways (languages, accents), but we can refer to them all as "talking humans." This simplifies code by allowing us to treat various objects uniformly when they share common behaviors. |
| | Polymorphism is the **ability to take on multiple forms**. Consider "Nayan" as a polymorphic speaker. You can speak in various languages, accents, and tones, but we can refer to you as a "speaking person" in a generalized way. |
| **Abstraction** | Abstraction is like **focusing on essential details**. When we talk about a "human," we don't need to know all their internal complexities. We abstract away the low-level details and focus on what's important: attributes like "name" and behaviors like "walking." |
| | Abstraction is **focusing on essential aspects**. When we talk about "Nayan," we don't need to know every minute detail. We abstract away the low-level details and focus on what's relevant, like your name and actions. |
| **Composition** | Composition is **combining smaller parts** to create a whole. Think of a "human" composed of body parts like "head," "torso," and "limbs." These parts are objects themselves, and they are combined to create a complete "human" object. |
| | Composition is **combining smaller elements** to create a whole. "Nayan" can be composed of body parts like "hands," "legs," and "head." These parts are objects themselves, and they are combined to create the complete "Nayan" object. |

| | |
|---|---|
| **Association** | Association **represents relationships** between humans. For instance, humans can have relationships like "friends" or "family members." These relationships are associations between individual humans (objects) that can be modeled in code. |
| | Association **represents relationships** with others. "Nayan" can have associations with "friends" and "family members." These relationships are associations between you (the "Nayan" object) and other individuals. |
| **Aggregation** | Aggregation is like humans living in houses. A "house" is composed of rooms, but it doesn't own them. Similarly, a "human" object may be composed of objects like "hands" or "legs," but these parts can exist independently and can be shared among different humans. |
| | Aggregation is like "Nayan" owning belongings. You may own objects like a **"phone," a "car," or a "laptop." These objects are part of your life, but they can exist independently** and may even be shared with others. |
| **Destructor** | A "human" may have a "destructor" action, like "dying" or "ceasing to exist." When a human object's lifetime ends, it goes through a destructor process, cleaning up resources or performing final actions. |
| | A "destructor" in your context could be something like "leaving a room." When you leave a room, you clean up, turn off the lights, and exit. It's a process that occurs when you're done with a particular task or environment. |
| **Friend Function** | Think of a "friend" as someone who can **access a human's private information**. In the context of a "human" class, a "friend function" could be a special function or entity that is granted access to a human object's private data or actions, just like a close friend might know personal details about you. |
| | Think of a "friend" as someone who knows **personal things about you**. In your context, a "friend function" could be a special function that is allowed to **access your private information**, like your full name or your personal preferences, just like a close friend might know these details about you. |

| | |
|---|---|
| **Class** is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated. | class Person {<br>public:<br>string name; int age;<br>Person(); // Constructor<br>Person(string personName, int personAge);<br>void displayInfo();<br>private:<br>string secret; // Private data member<br>void revealSecret();// Private member function<br>}; |
| **Object** is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions. | Person p = new Person(); |

Access specifiers are keywords that define how the **members of a class can be accessed** by other parts of the program. They are an important aspect of object-oriented programming, as they allow the programmer to control the **level of encapsulation and abstraction** of the class. There are three access specifiers in C++: **public, private**, and **protected**.

**Public:** The members declared as public can be **accessed by anyone**, including the objects of the **class**, the **functions outside the class**, and the **derived classes**. Public members are the interface of the class, as they expose the functionality and data that the class provides to the outside world.

**Private**: The members declared as private can only be accessed by the **member functions** and the **friend functions** of the class. They are **not accessible** by the objects of the class, the functions outside the class, or the derived classes. Private members are the implementation details of the class, as they hide the internal data and logic that the class uses to perform its tasks.

**Protected:** The members declared as protected can be accessed by the **member functions** of the class, the **friend functions** of the class, and the **member functions of the derived classes**. They are **not accessible** by the objects of the class or the functions outside the class. Protected members are **similar to private members**, but they **allow the inheritance** of data and methods by the subclasses.

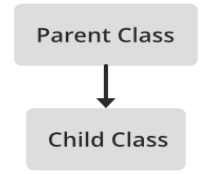
An **interface** is a concept in object-oriented programming that defines **a set of methods that a class must implement**. It provides a standard way to **define** the **behaviour** of a group of **related classes**. Interfaces allow you to specify what methods a class should implement, without providing the actual code for those methods. Interfaces make it easy to use a variety of different classes in the same way, as long as they implement the same interface. This is also known as **polymorphism**, which means having many forms of the same thing.

For example, suppose you have an interface called Animal, which defines a method called makeSound(). Any class that implements the Animal interface must provide a code for the makeSound() method. You can have different classes, such as Dog, Cat, and Bird, that implement the Animal interface and provide different sounds for the makeSound() method. Then, you can use an array or a list of type Animal to store objects of different classes that implement the Animal interface, and call the makeSound() method on each element of the array or list, without knowing the specific type of each object.

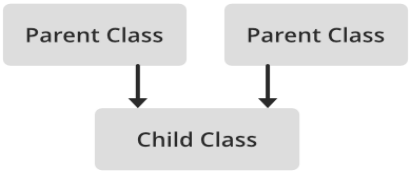
```
// Declare an interface
public interface Animal {
  // Define a method signature
  void makeSound();
}

// Declare a class that implements the interface
public class Dog implements Animal {
  // Provide the code for the method
  public void makeSound() {
    System.out.println("Woof");
  }
}
// Declare another class that implements the interface
public class Cat implements Animal {
  // Provide the code for the method
  public void makeSound() {
    System.out.println("Meow");
  }
}
// Declare a main class to test the interface
public class Main {
  public static void main
```

**Note**: When an object is created using a new keyword, then space is allocated for the variable in a **heap**, and the starting address is stored in the **stack** memory. When an object is created without a new keyword, then space is not allocated in the **heap** memory, and the object contains the **null value** in the **stack**.

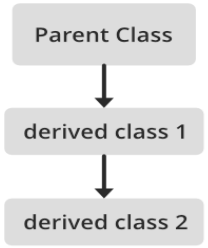
1. **Inheritance:**
   Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP) that allows you to create a new class (derived or subclass) based on an existing class (base or superclass). Inheritance establishes an "is-a" relationship between the new class and the existing class, where the derived class inherits the attributes and behaviors of the base class and can also add its own attributes and behaviors.



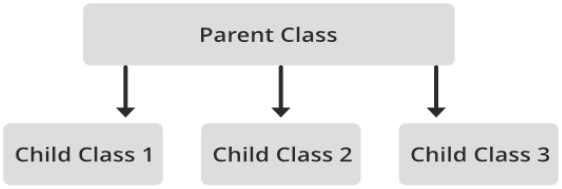
Single inheritance



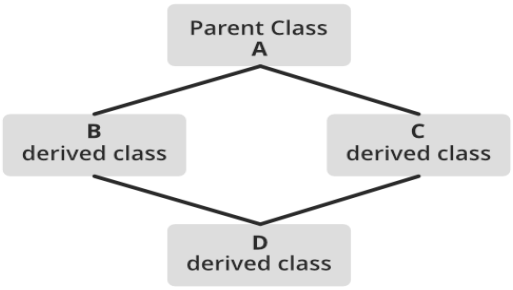
Multiple inheritance



Multilevel inheritance



Hierarchical inheritance



Hybrid Class

**Diamond problem** in inheritance:

In case of multiple inheritance, suppose class A has two subclasses B and C, and a class D has two super classes B and C. If a method present in A is overridden by both B and C but not by D then from which class D will inherit that method B or C? This problem is known as diamond problem.

| | |
|---|---|
| **Single Inheritance:**<br>• In single inheritance, a derived class inherits from a single base class.<br>• It's a simple form of inheritance and is commonly used in many programming languages, including C++, Java, and Python. | class Base {<br>  // Base class members<br>};<br><br>class Derived : public Base {<br>  // Derived class members<br>}; |
| **Multiple Inheritance:**<br>• In multiple inheritance, a derived class can inherit from more than one base class.<br>• This allows a class to inherit attributes and behaviors from multiple parent classes.<br>• Multiple inheritance can lead to issues like the "diamond problem," where ambiguity may arise if two base classes have a common method or attribute. | class Base1 {<br>  // Base class members<br>};<br>class Base2 {<br>  // Base class members<br>};<br>class Derived : public Base1, public Base2 {<br>  // Derived class members<br>}; |

| | |
|---|---|
| **Multilevel Inheritance:**<br>• In multilevel inheritance, a derived class is created from another derived class.<br>• It forms a chain of inheritance, where each derived class inherits from the one above it. | class Grandparent {<br>// Grandparent class members };<br>class Parent : public Grandparent {<br>// Parent class members };<br>class Child : public Parent {<br>// Child class members }; |
| **Hierarchical Inheritance:**<br>• In hierarchical inheritance, multiple derived classes inherit from a single base class.<br>• Each derived class can add its own attributes and behaviors while sharing the common attributes and behaviors from the base class. | class Animal {<br>// Base class members<br>};<br>class Dog : public Animal {<br>// Dog class members };<br>class Cat : public Animal {<br>// Cat class members }; |
| **Hybrid (or Mixed) Inheritance:**<br>• Hybrid inheritance is a combination of two or more types of inheritance.<br>• It can involve any combination of single, multiple, multilevel, or hierarchical inheritance. | class A {// Base class members};<br>class B : public A {// Derived class members};<br>class C { // Base class members };<br>class D : public B, public C { // Derived class members }; |

2. **Encapsulation**

Encapsulation is defined as the **wrapping up** of data under a **single unit**. It is the mechanism that **binds** together **code** and the **data** it manipulates. In Encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of their class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section, etc. 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. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Encapsulation can also be defined in two different ways:

- **Data hiding**: Encapsulation is the process of hiding unwanted information, such as restricting access to any member of an object.
- **Data binding**: Encapsulation is the process of binding the data members and the methods together as a whole, as a class.

**Types of Encapsulation in C++:**

| | |
|---|---|
| **Basic Encapsulation**:<br>• In basic encapsulation, you mark data members as private and provide public member functions to access and manipulate those data members.<br>• This is the most common form of encapsulation and is used to ensure data integrity and prevent unauthorized access. | class MyClass {<br>private:<br>int privateData;<br>public:<br>void setPrivateData(int value) {<br>// Validation and setting privateData<br>privateData = value;}<br>int getPrivateData() {<br>// Retrieve privateData<br>return privateData;}}; |
| **Getter and Setter Methods**:<br>• Getter methods (accessors) are used to retrieve the value of a private data member.<br>• Setter methods (mutators) are used to set the value of a private data member.<br>• This approach allows you to control access to the data and perform validation if needed. | |
| **Friend Functions and Classes**:<br>• In some cases, you may want to allow specific functions or classes to access private members of a class.<br>• You can declare them as friends using the **friend** keyword.<br>• This breaks encapsulation to some extent, so use it judiciously. | class MyClass {<br>private:<br>int privateData;<br>public:<br>friend void friendFunction(MyClass& obj);<br>};<br>void friendFunction(MyClass& obj) {<br>obj.privateData = 42; } |

## 3. Abstraction

We try to obtain an abstract view, model or structure of a real life problem, and reduce its unnecessary details. With definition of properties of problems, including the data which are affected and the operations which are identified, the model abstracted from problems can be a standard solution to this type of problems. It is an efficient way since there are nebulous real-life problems that have similar properties.

Abstraction is a concept of object-oriented programming that hides unnecessary details and shows only essential attributes. Abstraction allows developers to handle complexity by focusing on the desired behaviour rather than the implementation[13]. Abstraction is achieved by creating classes or interfaces that define the abstract properties and methods of the objects.

❖ **Abstraction in the real world**

I'm a coffee addict. So, when I wake up in the morning, I go into my kitchen, switch on the coffee machine and make coffee. Sounds familiar?
Making coffee with a coffee machine is a good example of abstraction.
You need to know how to use your coffee machine to make coffee. You need to provide water and coffee beans, switch it on and select the kind of coffee you want to get.
The thing you don't need to know is how the coffee machine is working internally to brew a fresh cup of delicious coffee. You don't need to know the ideal temperature of the water or the amount of ground coffee you need to use.
Someone else worried about that and created a coffee machine that now acts as an abstraction and hides all these details. You just interact with a simple interface that doesn't require any knowledge about the internal implementation.

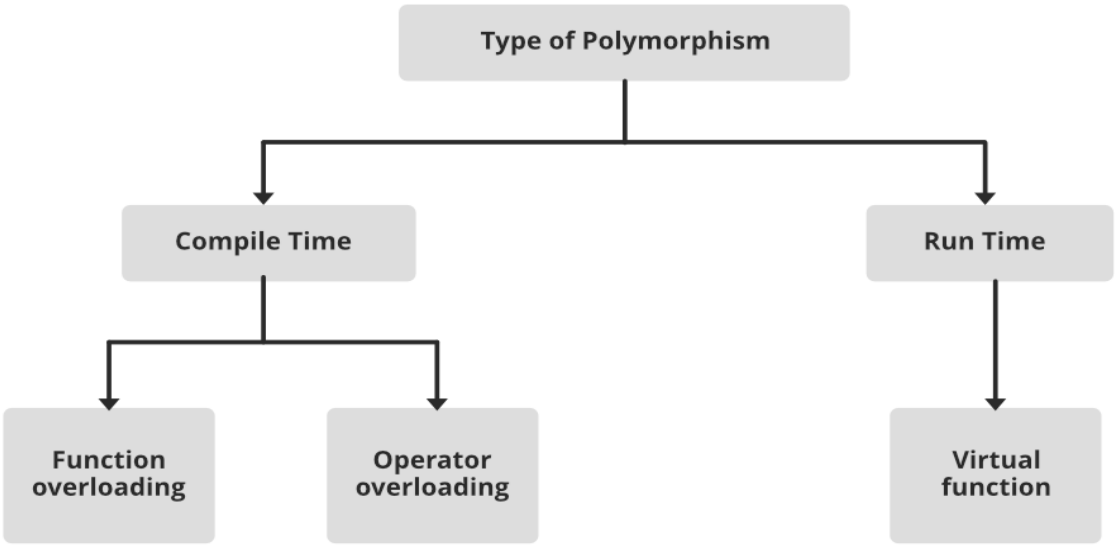
**Types of Abstraction in C++:**

| | |
|---|---|
| **Pure Abstract Class (Interface)**:<br>• A pure abstract class is a class that cannot be instantiated on its own. It defines a contract (an interface) for derived classes to implement.<br>• It contains only pure virtual functions (functions with no implementation) that derived classes must override. | class AbstractShape {<br>public:<br>virtual double area() const = 0;<br>// Pure virtual function<br>virtual void draw() const = 0;<br>// Pure virtual function<br>}; |
| **Abstract Base Class**:<br>• An abstract base class is similar to a pure abstract class but may contain a mix of pure virtual functions and regular member functions with implementations.<br>• It is still not meant to be instantiated directly; it serves as a common base for related classes. | class Animal {<br>public:<br>virtual void speak() const = 0;<br>// Pure virtual function<br>void eat() const {<br>// Implementation<br>}<br>}; |
| **Class Abstraction**:<br>• Class abstraction involves creating classes with a well-defined public interface and encapsulating data members as private.<br>• Users of the class interact with objects through the public member functions, while the internal details are hidden. | class BankAccount {<br>public:<br>void deposit(double amount);<br>void withdraw(double amount);<br>double getBalance() const;<br>private:<br>double balance;<br>}; |

4. **Polymorphism**

Polymorphism is the ability to present the same interface for differing underlying forms (data types). With polymorphism, each of these classes will have different underlying data. A point shape needs only two coordinates (assuming it's in a two-dimensional space of course). A circle needs a center and radius. A square or rectangle needs two coordinates for the top left and

bottom right corners and (possibly) a rotation. An irregular polygon needs a series of lines. Precisely, Poly means ‘many’ and morphism means ‘forms’.




**Working Process of Polymorphism:**

| | |
|---|---|
| **Function Overriding**:<br>• Function overriding is the process of providing a specific implementation of a base class's **virtual function** in a derived class.<br>• When a derived class overrides a virtual function, it replaces the behavior of the base class's version of that function.<br>• The overridden function in the derived class must have the same function signature (name, parameters, and return type) as the virtual function in the base class.<br>• To indicate that a function in the base class is meant to be overridden, you declare it as **virtual** in the base class. | class Shape {<br>public:<br>virtual void draw() const {<br>// Base class implementation<br>}<br>};<br><br>class Circle : public Shape {<br>public:<br>void draw() const override {<br>// Derived class implementation<br>}<br>}; |
| **Function Overloading**:<br>• Function overloading allows you to define multiple functions with the same name in a class, but with different parameter lists.<br>• The appropriate function to call is determined at compile-time based on the number and types of arguments provided.<br>• Function overloading is a form of compile-time (static) polymorphism. | class Calculator {<br>public:<br>int add(int a, int b) {<br>return a + b;<br>}<br>double add(double a, double b) {<br>return a + b;<br>}<br>}; |

### Method/Function Overloading:

Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

- The return type of the overloaded function.
- The type of the parameters passed to the function.
- The number of parameters passed to the function.

**Function Overriding:**

Function overriding means when the child class contains the method which is already present in the parent class. Hence, the child class overrides the method of the parent class. In case of function overriding, parent and child classes both contain the same function with a different definition.

**Types of Polymorphism in C++:**

| | |
|---|---|
| **Compile-Time Polymorphism (Static Polymorphism)**:<br>• Compile-time polymorphism, also known as **static polymorphism**, is **resolved at compile time**.<br>• It is achieved through function **overloading and operator overloading**.<br>• The appropriate function or operator to call is determined based on the number and types of arguments provided during compilation. | int add(int a, int b) {<br>  return a + b;<br>}<br><br>double add(double a, double b) {<br>  return a + b;<br>} |
| **Run-Time Polymorphism (Dynamic Polymorphism)**:<br>• Run-time polymorphism, also known as **dynamic polymorphism**, is resolved at **runtime**.<br>• It is achieved through **function overriding** and **virtual functions**.<br>• The appropriate function to call is determined based on the actual type of the object during runtime. | class Shape {<br>public:<br>  virtual void draw() const {<br>  // Base class implementation<br>  }};<br>class Circle : public Shape {<br>public:<br>  void draw() const override {<br>  // Derived class implementation}}; |

**Constructor:**

Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure.

- Constructor Name should be the same as a class name.
- A constructor must have no return type.

| | |
|---|---|
| **Default Constructor**:<br>• A default constructor takes no arguments and is automatically called when an object is created without any constructor arguments. | class MyClass {<br>public:<br>  MyClass() {<br>  // Default constructor implementation<br>  }<br>}; |

| | |
|---|---|
| • If you don't define any constructors in your class, C++ provides a default constructor automatically.<br>• It is invoked at the time of creating an object. | |
| **Parameterized Constructor**:<br>• A parameterized constructor accepts one or more arguments, allowing you to initialize the object's attributes with specific values when the object is created.<br>• It is used to provide different values to distinct objects. | class Student {<br>public:<br>Student(std::string name, int age) {<br>this->name = name;<br>this->age = age;<br>}<br>private:<br>std::string name;<br>int age;<br>}; |
| **Copy Constructor**:<br>• A copy constructor is used to create a new object as a copy of an existing object of the same class.<br>• It is called when an object is initialized with another object of the same type.<br>• A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. It is of two types –<br>• default copy constructor<br>• user defined copy constructor | class ComplexNumber {<br>public:<br>ComplexNumber(const ComplexNumber& other) {<br>// Copy constructor implementation<br>real = other.real;<br>imaginary = other.imaginary;<br>}<br>private:<br>double real;<br>double imaginary;}; |

**constructor overloading** is possible in object-oriented programming (OOP). Constructor overloading **allows** you to define **multiple** constructors within a class, each with a **different** set of **parameters**. The choice of constructor to be invoked **depends** on the **arguments** provided when an object is created.

**constructor overriding** is not a concept in object-oriented programming (OOP). In OOP, constructors are **not inherited** like regular member functions, and they cannot be overridden in the same way that virtual functions are overridden.

**Function Overloading** refers to the ability to define **multiple functions** in the same class or scope with the **same name** but different **parameters**. The functions must have different parameter lists, which can differ in terms of the number or types of parameters. When you call an overloaded function, the **compiler** determines which **version** of the function to **invoke** based on the number and types of **arguments** you **provide**.

**Function Overriding** is a feature that allows a subclass (or derived class) to provide a **specific implementation** for a function that is already defined in its superclass (or base class). The overridden function in the derived class should have the **same name**, **return** type, and **parameters** as the function in the base class. This allows you to provide a specialized implementation for a function in a subclass while maintaining a consistent interface.

| | |
|---|---|
| **Destructor:**<br>A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically. A destructor is defined like a constructor. It must have the same name as class, prefixed with a tilde sign (~). | class MyClass {<br>public:<br>  // Destructor<br>  ~MyClass() {<br>  // Destructor implementation<br>  }<br>}; |

**Key points about destructors:**

- Destructors are automatically called when an object goes out of scope, is explicitly deleted, or is part of an object allocated on the heap (created with **new**).
- Destructors are typically used to release any resources acquired by the object during its lifetime, such as freeing dynamically allocated memory, closing files, releasing network connections, or cleaning up any other resources used by the object.
- If you don't provide a custom destructor, C++ will generate a default destructor for your class, which essentially does nothing in terms of resource cleanup. However, when you have resources that need explicit cleanup, it's essential to provide a custom destructor.

‘**this**’ Pointer:
This is a keyword that refers to the **current instance of the class**. There can be 3 main uses of ‘this’ keyword:

- It can be used to pass the current object as a parameter to another method
- It can be used to refer to the current class instance variable.
- It can be used to declare indexers.

**Static Function:**
A static function in OOP is a function that **belongs to a class**, but **does not require** an **instance** of that **class** to be invoked. Static functions can **access** and **modify static variables**, which are variables that are shared by all instances of the class. Static functions are useful when they provide functionality that is related to the class as a whole, rather than to a specific object of that class. For example, a static function can act as a factory method that creates and returns objects of the class, or as a utility method that performs some common operation on the class or its objects.
you can call a static function on the class itself, without needing to **create** an **instance** of the **class**. Static functions are **associated** with the **class** rather than with individual **objects**.

**Static Function Code:**

```
#include <iostream>
class MyClass {
public:
  // Static function to calculate the square of a number
  static int square(int num) {
    return num * num;
  }
};
int main() {
  // Call the static function without creating an instance
  int result = MyClass::square(5);
  cout << "Square of 5 is: " << result << endl;
```

```
    return 0;
}
```

**Friend Function:**

Friend function acts as a friend of the class. It can access the **private** and **protected** members of the class. The friend function is **not a member of the class**, but it must be **listed in the class definition**. The non-member function cannot access the private data of the class. Sometimes, it is necessary for the non-member function to access the data. The friend function is a **non-member function** and has the ability to **access** the **private** data of the class.

**Note**:

- A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
- Friend function uses objects as arguments.

```
#include <iostream>
class MyClass;
void friendFunction(const MyClass& obj);
class MyClass {
private:
    int privateData;
public:
    MyClass(int data) : privateData(data) {}
friend void friendFunction(const MyClass& obj);
};
void friendFunction(const MyClass& obj) {
    cout << "Friend Function accessed privateData: " << obj.privateData << endl;
}
int main() {
    MyClass myObject(42);
    friendFunction(myObject);
    return 0;
}
```

**Virtual Function:**

A virtual function is a function that can be **overridden** by a derived class to provide **different behavior** for the same function name. Virtual functions are used to achieve **runtime polymorphism**, which means that the correct function to be executed is determined at runtime based on the actual type of the object. Virtual functions are declared with the **virtual** keyword in the base class and can be redefined in the derived class without the **virtual** keyword.

For **example**, consider a base class Animal and a derived class Dog. The base class has a virtual function speak() that prints “I am an animal”. The derived class overrides this function and prints “I am a dog”. If we have a pointer of type Animal that points to an object of type Dog, then calling the speak() function on this pointer will

**Virtual Function Code:**

```
#include <iostream>

class Animal {
public:
    virtual void makeSound() {
        cout << "Animal makes a generic sound" << endl;
    }
};
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Dog barks" << endl;
    }
};
class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Cat meows" << endl;
    }
};
int main() {
    Animal* animalPtr;
```

invoke the Dog's version of the function, not the Animal's version. This is because the function is resolved at runtime based on the actual type of the object, not the declared type of the pointer.

**Key Points:**

1. Virtual functions cannot be static.
2. A class may have a virtual destructor but it cannot have a virtual constructor.

```
    Dog dog;
    Cat cat;
    animalPtr = &dog;
    animalPtr->makeSound();      // Calls Dog's implementation
    animalPtr = &cat;
    animalPtr->makeSound();      // Calls Cat's implementation

    return 0;
}
```

A **pure virtual function** in OOP is a function that is declared in a base class but has no definition in the base class. It is marked with "=0" to indicate that it is pure. A pure virtual function must be **overridden** by a derived class that inherits from the base class. A class that contains a **pure virtual function** is called an **abstract class** and cannot be instantiated. A pure virtual function is used to achieve **runtime polymorphism**, which means that different classes can have different behaviors for the same function name.

Pure virtual functions are also referred to as **abstract methods**. The key distinction between a pure virtual function and a regular virtual function is that a pure virtual function has **no default implementation** in the base class.

```
#include <iostream>
class Shape {
public:
    virtual void draw() const = 0;
};
// Derived class 1
class Circle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a circle." << endl;
    }
};
// Derived class 2
class Rectangle : public Shape {
public:
    void draw() const override {
        cout << "Drawing a rectangle." <<endl;
    }
};
int main() {
    Circle circle;
    Rectangle rectangle;
    const Shape* shapePtr = &circle;
    shapePtr->draw();
    shapePtr = &rectangle;
    shapePtr->draw();
    return 0;
}
```

**Abstract Method:**

An abstract method is a method declared in an **abstract class** but does not have a concrete (implemented)

```
#include <iostream>
class Shape {
public:
    virtual void draw() const = 0;
```

| | |
|---|---|
| definition in the abstract class itself. Instead, it is meant to be overridden and implemented by any concrete (non-abstract) subclass of the abstract class. Abstract methods define a contract that derived classes must adhere to, ensuring that they provide specific functionality.<br><br>Abstract methods are methods that are declared in an abstract class or an interface, but not defined. They act as placeholders for the subclasses or the implementing classes to provide their own definitions. Abstract methods are used to specify the common behavior or contract that the subclasses or the implementing classes must follow. | };<br>// Derived class 1<br>class Circle : public Shape {<br>public:<br>  void draw() const override {<br>  cout << "Drawing a circle." << endl;<br>  }<br>};<br>// Derived class 2<br>class Rectangle : public Shape {<br>public:<br>  void draw() const override {<br>  cout << "Drawing a rectangle." <<endl;<br>  }<br>};<br>int main() {<br>  Circle circle;<br>  Rectangle rectangle;<br>  const Shape* shapePtr = &circle;<br>  shapePtr->draw();<br>  shapePtr = &rectangle;<br>  shapePtr->draw();<br>  return 0;<br>} |
| **Abstract Class:**<br>An abstract class in Object-Oriented Programming (OOP) is a class that cannot be **instantiated**, meaning that you **cannot create** an **object** using the abstract class. Instead, an abstract class is **used** as a **base** for other classes that inherit from it and provide the implementation of its **abstract methods**. Abstract classes are useful for modelling concepts or behaviours that are common to a group of subclasses, but may vary in details.<br><br>For **example**, you can have an abstract class called Animal that defines some properties and methods that all animals have, such as name, age, eat, sleep, etc. However, you cannot create an object of type Animal, because it is too general and does not specify how each animal eats or sleeps. You can create subclasses | #include <iostream><br>class Shape {<br>public:<br>  virtual double calculateArea() const = 0;<br>  void printArea() const {<br>  cout << "Area: " << calculateArea() << endl;<br>  }<br>};<br>// Derived class 1<br>class Circle : public Shape {<br>private:<br>  double radius;<br>public:<br>  Circle(double r) : radius(r) {}<br>  // Implement the abstract method<br>  double calculateArea() const override {<br>  return 3.14159265359 * radius * radius;<br>  }<br>};<br>// Derived class 2<br>class Rectangle : public Shape {<br>private:<br>  double width; |

of Animal, such as Dog, Cat, Bird, etc., that inherit from the Animal class and provide their own implementation of the abstract methods. This way, you can reuse the code from the Animal class and avoid duplication.

In C++ class is made abstract by declaring at least **one** of its **functions** as a **pure virtual function**. A pure virtual function is specified by placing "=0" in its declaration. Its implementation must be provided by derived classes.

```
    double height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    // Implement the abstract method
    double calculateArea() const override {
        return width * height;
    }
};
int main() {
    Circle circle(5.0);
    Rectangle rectangle(4.0, 6.0);
    // Using polymorphism to treat objects of different derived classes as Shape objects
    Shape* shapePtr = &circle;
    shapePtr->printArea();
    shapePtr = &rectangle;
    shapePtr->printArea();
    return 0;
}
```

**Namespaces in C++:**

The main purpose of using namespace in C++ is to **remove** the **ambiguity**. Ambiguity occurs when a different task occurs with the **same name**.

1. Avoiding Name Collisions
2. Organization and Structure
3. Separation of Concerns
4. Avoiding Global Scope Pollution
5. Library Organization
6. Versioning and Compatibility
7. Readability and Documentation
8. Selective Importing

**Association**:

Association is a relationship between two or more classes that describes how objects of these classes are connected or interact with each other. Association represents a "using" or "working together" relationship between classes, where one class uses or is related to another class for some purpose.

**Aggregation**:

Aggregation is a type of **association** that represents a "whole-part" relationship between classes, where one class (the whole) contains or is composed of one or more objects of another class (the part). Aggregation implies a stronger relationship than a simple association, as it suggests that one class is composed of or owns the other class or classes. The objects that are part of the whole class can exist independently, even if they are removed from the whole.

Aggregation is also known as "HAS-A" relationship.

**Composition:**

Composition is a type of **association** that represents a "whole-part" relationship between classes, where one class (the whole) is composed of or contains one or more objects of another class (the part). Composition implies a strong ownership relationship, where the part objects are considered an integral part of the whole object. In other words, the lifetime of the part objects is tightly coupled to the lifetime of the whole object.

- Composition is a special form of Aggregation where the part cannot exist without the whole.
- Composition is a strong Association.
- Composition relationship is represented like aggregation with one difference that the diamond shape is filled.