**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 blueprint or template for creating objects. It defines the structure and behavior that objects of that class will have. Classes encapsulate data and methods, providing a blueprint for creating multiple instances (objects) with the same structure and behavior. |
| **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 concept of bundling data (attributes) and the methods (functions) that operate on that data within a single unit, the class. It hides the internal details of how an object works and provides a well-defined interface for interacting with it. This helps in data protection and prevents unauthorized access or modification of an object's internal state. |
| **Abstraction**: Abstraction is the process of simplifying complex systems by modeling them with a high-level representation that focuses on essential characteristics while hiding unnecessary details. | **Polymorphism**: Polymorphism is nothing but assigning behavior or value in a subclass to something that was already declared in the main class. Simply, polymorphism takes more than one form |

|  |  |
| --- | --- |
| **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 {  public:  string name; int age;  Person(); // Constructor  Person(string personName, int personAge);  void displayInfo();  private:  string secret; // Private data member  void revealSecret();// Private member function  }; |
| **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(); |

**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.

**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:**   * In single inheritance, a derived class inherits from a single base class. * It's a simple form of inheritance and is commonly used in many programming languages, including C++, Java, and Python. | class Base {  // Base class members  };  class Derived : public Base {  // Derived class members  }; |
| **Multiple Inheritance:**   * In multiple inheritance, a derived class can inherit from more than one base class. * This allows a class to inherit attributes and behaviors from multiple parent classes. * 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 {  // Base class members  };  class Base2 {  // Base class members  };  class Derived : public Base1, public Base2 {  // Derived class members  }; |
| **Multilevel Inheritance:**   * In multilevel inheritance, a derived class is created from another derived class. * It forms a chain of inheritance, where each derived class inherits from the one above it. | class Grandparent {  // Grandparent class members  };  class Parent : public Grandparent {  // Parent class members  };  class Child : public Parent {  // Child class members  }; |
| **Hierarchical Inheritance:**   * In hierarchical inheritance, multiple derived classes inherit from a single base class. * Each derived class can add its own attributes and behaviors while sharing the common attributes and behaviors from the base class. | class Animal {  // Base class members  };  class Dog : public Animal {  // Dog class members  };  class Cat : public Animal {  // Cat class members  }; |
| **Hybrid (or Mixed) Inheritance:**   * Hybrid inheritance is a combination of two or more types of inheritance. * It can involve any combination of single, multiple, multilevel, or hierarchical inheritance. | class A {// Base class members};  class B : public A {// Derived class members};  class C { // Base class members };  class D : public B, public C { // Derived class members }; |

1. **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.

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

1. **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](https://www.bing.com/ck/a?!&&p=54a5e016ac796e14JmltdHM9MTY5NDIxNzYwMCZpZ3VpZD0zMDMwOGM0NS1jYTEyLTYxMjctMzAxMC05ZTQxY2IxMzYwZTkmaW5zaWQ9NTc5MQ&ptn=3&hsh=3&fclid=30308c45-ca12-6127-3010-9e41cb1360e9&psq=what+is+abstraction+in+oop&u=a1aHR0cHM6Ly9zdGFja2lmeS5jb20vb29wLWNvbmNlcHQtYWJzdHJhY3Rpb24v&ntb=1). [Abstraction allows developers to handle complexity by focusing on the desired behaviour rather than the implementation](https://www.bing.com/ck/a?!&&p=303b34dfaf40d148JmltdHM9MTY5NDIxNzYwMCZpZ3VpZD0zMDMwOGM0NS1jYTEyLTYxMjctMzAxMC05ZTQxY2IxMzYwZTkmaW5zaWQ9NTc5NA&ptn=3&hsh=3&fclid=30308c45-ca12-6127-3010-9e41cb1360e9&psq=what+is+abstraction+in+oop&u=a1aHR0cHM6Ly9zdGFja2lmeS5jb20vb29wLWNvbmNlcHQtYWJzdHJhY3Rpb24v&ntb=1)[1](https://www.bing.com/ck/a?!&&p=95612befe9fd4f0fJmltdHM9MTY5NDIxNzYwMCZpZ3VpZD0zMDMwOGM0NS1jYTEyLTYxMjctMzAxMC05ZTQxY2IxMzYwZTkmaW5zaWQ9NTc5NQ&ptn=3&hsh=3&fclid=30308c45-ca12-6127-3010-9e41cb1360e9&psq=what+is+abstraction+in+oop&u=a1aHR0cHM6Ly9zdGFja2lmeS5jb20vb29wLWNvbmNlcHQtYWJzdHJhY3Rpb24v&ntb=1)[3](https://www.bing.com/ck/a?!&&p=e1659e2ff61c677cJmltdHM9MTY5NDIxNzYwMCZpZ3VpZD0zMDMwOGM0NS1jYTEyLTYxMjctMzAxMC05ZTQxY2IxMzYwZTkmaW5zaWQ9NTc5Ng&ptn=3&hsh=3&fclid=30308c45-ca12-6127-3010-9e41cb1360e9&psq=what+is+abstraction+in+oop&u=a1aHR0cHM6Ly9wYWxpemdhci5tZWRpdW0uY29tL2Fic3RyYWN0aW9uLWluLW9iamVjdC1vcmllbnRlZC1wcm9ncmFtbWluZy1iZW5lZml0cy1hbmQtZXhhbXBsZXMtYjBiZWZjZGNmN2Qw&ntb=1). 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)**:   * 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. * It contains only pure virtual functions (functions with no implementation) that derived classes must override. | class AbstractShape {  public:  virtual double area() const = 0;  // Pure virtual function  virtual void draw() const = 0;  // Pure virtual function  }; |
| **Abstract Base Class**:   * 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. * It is still not meant to be instantiated directly; it serves as a common base for related classes. | class Animal {  public:  virtual void speak() const = 0;  // Pure virtual function  void eat() const {  // Implementation  }  }; |
| **Class Abstraction**:   * Class abstraction involves creating classes with a well-defined public interface and encapsulating data members as private. * Users of the class interact with objects through the public member functions, while the internal details are hidden. | class BankAccount {  public:  void deposit(double amount);  void withdraw(double amount);  double getBalance() const;  private:  double balance;  }; |

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

* **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)**:   * Compile-time polymorphism, also known as **static polymorphism**, is **resolved at compile time**. * It is achieved through function **overloading and operator overloading**. * 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) {  return a + b;  }  double add(double a, double b) {  return a + b;  } |
| **Run-Time Polymorphism (Dynamic Polymorphism)**:   * Run-time polymorphism, also known as **dynamic polymorphism**, is resolved at **runtime**. * It is achieved through **function overriding** and **virtual functions**. * The appropriate function to call is determined based on the actual type of the object during runtime. | class Shape {  public:  virtual void draw() const {  // Base class implementation  }};  class Circle : public Shape {  public:  void draw() const override {  // 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**:   * A default constructor takes no arguments and is automatically called when an object is created without any constructor arguments. * If you don't define any constructors in your class, C++ provides a default constructor automatically. * It is invoked at the time of creating an object. | class MyClass {  public:  MyClass() {  // Default constructor implementation  }  }; |
| **Parameterized Constructor**:   * A parameterized constructor accepts one or more arguments, allowing you to initialize the object's attributes with specific values when the object is created. * It is used to provide different values to distinct objects. | class Student {  public:  Student(std::string name, int age) {  this->name = name;  this->age = age;  }  private:  std::string name;  int age;  }; |
| **Copy Constructor**:   * A copy constructor is used to create a new object as a copy of an existing object of the same class. * It is called when an object is initialized with another object of the same type. * A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object. It is of two types –   + default copy constructor   + user defined copy constructor | class ComplexNumber {  public:  ComplexNumber(const ComplexNumber& other) {  // Copy constructor implementation  real = other.real;  imaginary = other.imaginary;  }  private:  double real;  double imaginary;  }; |

|  |  |
| --- | --- |
| **Destructor:**  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 {  public:  // Destructor  ~MyClass() {  // Destructor implementation  }  }; |

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.

|  |
| --- |
| **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. |

|  |  |
| --- | --- |
| **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 {  private:  int data; // Private data member  public:  MyClass(int value) : data(value) { }  void showData(const MyClass& obj);  };  // Define the friend function outside of the  void showData(const MyClass& obj) {  cout << "Data: " << obj.data << endl;  }  int main() {  MyClass obj(42);    // Call the friend function to access the private member  showData(obj);  return 0;  } |