OOPs

1. Class

A **class** in C++ is a blueprint or template for creating objects. It defines the properties (data members) and behaviors (methods or member functions) that objects of the class will have. Essentially, a class serves as a user-defined data type. By grouping related data and functions together, classes provide modularity and structure to a program.

The primary purpose of a class is to provide a way to model real-world entities, with each class encapsulating attributes and actions. For instance, a class `Car` could define properties like `make`, `model`, and `year`, and behaviors like `drive()` or `stop()`. Classes are fundamental to organizing and managing complex systems, ensuring that the data is logically grouped and can be manipulated in a controlled manner.

2. Object

An **object** is an instance of a class. When a class is defined, no memory is allocated for it; only when an object of the class is created does memory get allocated. Objects represent real-world entities and hold specific values for the properties defined in the class.

Objects allow you to work with actual instances of the abstract definitions provided by the class. Each object has its own copy of data members, meaning the values held by one object are independent of the others. Objects interact with each other and can invoke methods defined in the class to modify their state or perform actions.

For example, if `Car` is a class, then an object `myCar` is an instance of the class `Car`, holding its own specific values for attributes like `make`, `model`, and `year`.

3. Inheritance

Inheritance is a mechanism in C++ that allows one class (called the **derived class**) to inherit properties and behaviors from another class (called the **base class**). This facilitates code reuse and establishes a hierarchical relationship between classes.

In C++, inheritance allows the derived class to reuse the functionality of the base class and can also extend or modify that functionality. This results in less code duplication and a more maintainable codebase. Inheritance models the "is-a" relationship — for example, a `Dog` class might inherit from an `Animal` class, since a dog "is a" type of animal.

There are various types of inheritance in C++:

- **Single inheritance**: A class inherits from one base class.
- **Multiple inheritance**: A class inherits from more than one base class.
- **Multilevel inheritance**: A class inherits from a class that already inherits from another class.
- **Hierarchical inheritance**: Multiple classes inherit from a single base class.

Inheritance helps in building a well-organized class hierarchy and in promoting code reusability.

4. Polymorphism

Polymorphism in C++ allows one interface to be used for different data types or objects. It means that a single function, method, or operator can work in different ways depending on the object it is acting upon.

Polymorphism can be classified into:

- **Compile-time polymorphism (Static Polymorphism)**: This occurs when the function call is resolved at compile time. It is typically achieved through **function overloading** and **operator overloading**. For example, the same function name might perform different actions based on the type or number of arguments.

- **Runtime polymorphism (Dynamic Polymorphism)**: This occurs when the function call is resolved at runtime. It is achieved through **method overriding** and involves the use of **virtual functions**. In this case, a base class pointer can invoke methods from derived classes, ensuring that the correct method for the actual object type is called, even if the pointer is of the base class type.

Polymorphism allows a more flexible and extensible design, where different objects can be treated in a similar way through a common interface, making systems more maintainable and scalable.

5. Abstraction

Abstraction in C++ is the concept of hiding the complex implementation details and showing only the essential features of an object. It is the process of simplifying complex systems by exposing only the relevant information and hiding the internal workings.

Abstraction helps in reducing complexity by focusing on what an object does rather than how it does it. This is typically achieved using **abstract classes** and **interfaces**. An abstract class contains one or more **pure virtual functions**, which do not have implementations in the base class but must be implemented by derived classes.

By using abstraction, a programmer can interact with high-level functionality without needing to understand the intricate details of its implementation. This leads to cleaner and more understandable code.

6. Encapsulation

Encapsulation is the concept of bundling data (attributes) and the methods (functions) that operate on that data into a single unit called a **class**. In C++, encapsulation is achieved by controlling access to the data using **access specifiers** such as `private`, `protected`, and `public`.

Encapsulation also involves **data hiding**, meaning that the internal state of an object is hidden from the outside world, and can only be accessed or modified via methods (getters and setters). This prevents external code from directly manipulating the object's data in unintended ways and ensures that the object’s state remains consistent and valid.

Encapsulation makes software easier to maintain and modify by isolating the internal workings of objects. It also helps in protecting the data from unauthorized access and modification, leading to better security and stability in programs.
