ASSIGNMENT -3

1. 1. What is an object in C++?

An **object** in C++ is an instance of a class. It represents a real-world entity with attributes (data members) and behaviors (member functions). Objects are created from classes and can interact with each other through their methods.

Example:

```
cpp
CopyEdit
class Car {
public:
    void start() {
        cout << "Car started." << endl;
    }
};

int main() {
    Car myCar; // Object creation
    myCar.start(); // Calling method
    return 0;
}
```

2. What is a class in C++ and how does it differ from an object?

A **class** is a blueprint or template for creating objects. It defines the properties and behaviors that the objects created from it will have. An **object**, on the other hand, is a specific instance of a class.

Difference:

- **Class**: Defines structure and behavior.
- **Object**: An instance of the class with actual data.

3. Explain the concept of encapsulation with an example.

Encapsulation is the bundling of data and methods that operate on that data within a single unit (class), restricting direct access to some of an object's components.

Example:

```
cpp
CopyEdit
class Account {
private:
    double balance;
```

```
public:
    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    double getBalance() const {
        return balance;
    }
};
```
Here, `balance` is private, and access is controlled via public methods.

4. How do you define a class in C++?

A class in C++ is defined using the `class` keyword followed by the class name and its members.

Syntax:

```
cpp
CopyEdit
class ClassName {
private:
    // private members

public:
    // public members
};
```

5. Describe the syntax for creating an object of a class.

An object is created by declaring a variable of the class type.

Syntax:

```
cpp
CopyEdit
ClassName objectName;
```
Example:

```
cpp
CopyEdit
Car myCar;
```

6. What are private members in a class and how are they accessed?

Private members are variables and methods declared with the `private` access specifier. They cannot be accessed directly outside the class.

Accessing private members:

- Through **public methods** (getters/setters).
- Using **friend functions**.

7. What are public members in a class and how are they accessed?

Public members are variables and methods declared with the `public` access specifier. They can be accessed directly from outside the class.

Example:

```
cpp
CopyEdit
class Car {
public:
    void start() {
        cout << "Car started." << endl;
    }
};
```

8. Explain the significance of access specifiers in a class.

Access specifiers (`public`, `private`, `protected`) control the accessibility of class members:

- **public**: Accessible from anywhere.
- **private**: Accessible only within the class.
- **protected**: Accessible within the class and by derived classes.

9. Provide an example of a class with both private and public members.

```
cpp
CopyEdit
class Car {
private:
    string model;

public:
    void setModel(string m) {
        model = m;
    }

    string getModel() const {
```

```
        return model;
    }

    void start() {
        cout << model << " started." << endl;
    }
};
```

10. How does data hiding work in C++?

Data hiding is achieved by declaring class members as `private` or `protected`, preventing direct access from outside the class. This ensures that the internal state of an object cannot be changed arbitrarily.

11. What is a static data member in C++?

A **static data member** is shared by all objects of a class. It is initialized outside the class definition.

Example:

```
cpp
CopyEdit
class Car {
public:
    static int count;
};

int Car::count = 0;
```

12. How do you declare and initialize a static data member?

Declaration:

```
cpp
CopyEdit
static dataType memberName;
```

Initialization:

```
cpp
CopyEdit
dataType ClassName::memberName = value;
```

13. What is a static function member in C++?

A **static function member** is a function that can be called without creating an object of the class. It can only access static data members.

Example:

```
cpp
CopyEdit
class Car {
public:
    static void displayCount() {
        cout << "Total cars: " << count << endl;
    }
};
```

14. How do static function members differ from regular function members?

- **Static functions**: Can be called without an object, can access only static members.
- **Regular functions**: Require an object to be called, can access both static and non-static members.

15. Provide an example of a class with static data and function members.

```
cpp
CopyEdit
class Car {
public:
    static int count;

    Car() {
        count++;
    }

    static void displayCount() {
        cout << "Total cars: " << count << endl;
    }
};

int Car::count = 0;
```

16. What is a constructor in C++ and why is it important?

A **constructor** is a special member function that is called when an object is created. It initializes the object's members.

Importance:

- Ensures objects are initialized properly.
- Can allocate resources.

17. Explain the different types of constructors in C++.

- **Default constructor**: No parameters.
- **Parameterized constructor**: Takes parameters to initialize members.
- **Copy constructor**: Initializes a new object as a copy of an existing object.

18. What is a default constructor and when is it used?

A **default constructor** is a constructor that takes no arguments. It is used when an object is created without providing initial values.

Example:

```
cpp
CopyEdit
class Car {
public:
    Car() {
        // Initialization code
    }
};
```

19. How do parameterized constructors work?

A **parameterized constructor** takes arguments to initialize an object with specific values.

Example:

```
cpp
CopyEdit
class Car {
public:
    Car(string m) {
        model = m;
    }
};
```

20. What is a copy constructor and what is its purpose?

A **copy constructor** initializes a new object as a copy of an existing object. It is used when passing objects by value or returning objects from functions.

Example:

```
cpp
CopyEdit
class Car {
```

```
public:
    Car(const Car& other) {
        model = other.model;
    }
};
```

22. Constructor Initializer List

A **constructor initializer list** is used to initialize class members before the constructor body executes. It's particularly useful for initializing `const` members and base class constructors.

Syntax:

```
cpp
CopyEdit
ClassName::ClassName(parameters) : member1(value1),
member2(value2) {
    // Constructor body
}
```

Example:

```
cpp
CopyEdit
class Car {
private:
    const string model;
    int year;

public:
    Car(string m, int y) : model(m), year(y) {}
};
```

In this example, the `model` member is initialized using the initializer list.

23. Destructor in C++

A **destructor** is a special member function that is called when an object goes out of scope or is explicitly deleted. Its primary purpose is to release resources acquired by the object.

Syntax:

```
cpp
CopyEdit
~ClassName() {
    // Cleanup code
}
```

Example:

```
cpp
CopyEdit
class Car {
public:
    ~Car() {
        cout << "Car object destroyed." << endl;
    }
};
```
Here, the destructor outputs a message when a `Car` object is destroyed.

24. Declaring and Defining a Destructor

A destructor is declared inside the class definition and defined outside the class if needed.

Example:

```
cpp
CopyEdit
class Car {
public:
    ~Car(); // Declaration
};

Car::~Car() { // Definition
    cout << "Car object destroyed." << endl;
}
```

25. If Destructor is Not Explicitly Defined

If a destructor is not explicitly defined, the compiler provides a default destructor that performs no action. However, if the class manages resources like dynamic memory, it's crucial to define a destructor to release those resources.

26. Automatic vs. Dynamic Storage Duration

- **Automatic storage duration**: Objects are created when declared and destroyed when they go out of scope (e.g., local variables).
- **Dynamic storage duration**: Objects are created using `new` and must be destroyed using `delete`.

Destructors are automatically called for objects with automatic storage duration. For dynamically allocated objects, you must explicitly call `delete` to invoke the destructor.

27. Difference Between Constructors and Destructors

- **Constructors**: Initialize objects and allocate resources.
- **Destructors**: Clean up resources and perform necessary cleanup before object destruction.

28. Operator Overloading in C++

Operator overloading allows you to define custom behavior for operators (like `+`, `-`, `*`, etc.) when applied to objects of a class.

Syntax:

```
cpp
CopyEdit
ReturnType operator op (parameters) {
    // Implementation
}
```

Example:

```
cpp
CopyEdit
class Complex {
private:
    int real, imag;

public:
    Complex operator + (const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }
};
```

In this example, the `+` operator is overloaded to add two `Complex` numbers.

29. Syntax for Overloading an Operator

The syntax for overloading an operator is as follows:

```
cpp
CopyEdit
ReturnType operator op (parameters) {
    // Implementation
}
```

For example, overloading the `+` operator for a `Complex` class:

```
cpp
CopyEdit
Complex operator + (const Complex& other) {
    return Complex(real + other.real, imag + other.imag);
}
```

30. Operators That Can and Cannot Be Overloaded

Can be overloaded:

- Arithmetic operators (`+`, `-`, `*`, `/`)
- Comparison operators (`==`, `!=`, `<`, `>`)
- Assignment operator (`=`)
- Subscript operator (`[]`)
- Function call operator (`()`)
- Increment/Decrement operators (`++`, `--`)

Cannot be overloaded:

- Scope resolution operator (`::`)
- Member access operators (`.`, `.*`)
- Ternary conditional operator (`?:`)
- Typeid operator (`typeid`)
- Sizeof operator (`sizeof`)

31. Overloading the "+" Operator for a Custom Class

Example:

```
cpp
CopyEdit
class Complex {
private:
    int real, imag;

public:
    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator + (const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
```

```
    }
};
```
Here, the + operator is overloaded to add two `Complex` numbers.

32. Friend Functions in Operator Overloading

A **friend function** is a function that is not a member of a class but has access to its private and protected members. Friend functions can be used to overload operators that require access to private members.

33. Friend Function in C++ and How It's Declared

A friend function is declared inside the class definition using the `friend` keyword.

Syntax:

```
cpp
CopyEdit
friend ReturnType functionName(parameters);
```
Example:

```
cpp
CopyEdit
class Complex {
private:
    int real, imag;

public:
    friend Complex operator + (const Complex& c1, const
Complex& c2);
};
```

34. Difference Between Friend Functions and Member Functions

- **Member functions**: Defined inside the class, have access to all members (private, protected, public).
- **Friend functions**: Defined outside the class, but have access to all members due to the `friend` declaration.

35. Benefits and Drawbacks of Using Friend Functions

Benefits:

- Can access private and protected members of a class.

- Useful for operator overloading and non-member functions that need access to class internals.

Drawbacks:

- Breaks encapsulation by allowing external functions to access private data.
- Can lead to maintenance issues if overused.

36. Inheritance in C++

Inheritance is a mechanism where a new class (derived class) acquires the properties and behaviors of an existing class (base class). It's important for code reusability and establishing relationships between classes.

37. Types of Inheritance in C++

C++ supports various types of inheritance:

1. **Single Inheritance**: A derived class inherits from only one base class.
    - Example: `class Derived : public Base {};`
2. **Multiple Inheritance**: A derived class inherits from two or more base classes.
    - Example: `class Derived : public Base1, public Base2 {};`
3. **Multilevel Inheritance**: A class is derived from a class, which is itself derived from another class.
    - Example: `class Base {}; class Derived : public Base {}; class FurtherDerived : public Derived {};`
4. **Hierarchical Inheritance**: A single base class is inherited by multiple derived classes.
    - Example: `class Base {}; class Derived1 : public Base {}; class Derived2 : public Base {};`
5. **Hybrid Inheritance**: A combination of two or more types of inheritance, like multiple and multilevel inheritance.
    - Example: `class Base {}; class Derived1 : public Base {}; class Derived2 {}; class Final : public Derived1, public Derived2 {};`

38. Implementing Single Inheritance in C++

Single inheritance means a derived class inherits from one base class.

Example:

cpp
CopyEdit
```
class Animal {
public:
    void speak() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void speak() {
        cout << "Dog barks." << endl;
    }
};

int main() {
    Dog d;
    d.speak(); // Output: Dog barks.
    return 0;
}
```
Here, `Dog` inherits from the `Animal` class, and we override the `speak` function in the derived class.

39. Multiple Inheritance and How It Differs from Single Inheritance

Multiple inheritance occurs when a derived class inherits from more than one base class.

Example:

cpp
CopyEdit
```
class Base1 {
public:
    void displayBase1() {
        cout << "Base1 class" << endl;
    }
};

class Base2 {
```

```
public:
    void displayBase2() {
        cout << "Base2 class" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    void display() {
        cout << "Derived class" << endl;
    }
};

int main() {
    Derived obj;
    obj.displayBase1();  // Output: Base1 class
    obj.displayBase2();  // Output: Base2 class
    obj.display();       // Output: Derived class
    return 0;
}
```

Difference from Single Inheritance: In multiple inheritance, a derived class inherits from multiple base classes, whereas in single inheritance, it inherits from only one.

40. Hierarchical Inheritance with an Example

Hierarchical inheritance occurs when multiple derived classes inherit from a single base class.

Example:

```
cpp
CopyEdit
class Animal {
public:
    void sound() {
        cout << "Animal makes a sound." << endl;
    }
};

class Dog : public Animal {
public:
    void sound() {
        cout << "Dog barks." << endl;
    }
};

class Cat : public Animal {
```

```
public:
    void sound() {
        cout << "Cat meows." << endl;
    }
};

int main() {
    Dog dog;
    Cat cat;
    dog.sound();  // Output: Dog barks.
    cat.sound();  // Output: Cat meows.
    return 0;
}
```

In this example, `Dog` and `Cat` both inherit from `Animal`.

41. Multilevel Inheritance and How It's Implemented in C++

Multilevel inheritance occurs when a class is derived from another derived class.

Example:

```
cpp
CopyEdit
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};

class Mammal : public Animal {
public:
    void breathe() {
        cout << "Breathing..." << endl;
    }
};

class Dog : public Mammal {
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};

int main() {
```

```
    Dog d;
    d.eat();    // Output: Eating...
    d.breathe(); // Output: Breathing...
    d.bark();    // Output: Barking...
    return 0;
}
```

Here, `Dog` inherits from `Mammal`, and `Mammal` inherits from `Animal`.

42. Hybrid Inheritance

Hybrid inheritance is a combination of multiple inheritance types like multiple inheritance and multilevel inheritance.

Example:

```
cpp
CopyEdit
class Base1 {
public:
    void func1() {
        cout << "Base1" << endl;
    }
};

class Base2 {
public:
    void func2() {
        cout << "Base2" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    void func3() {
        cout << "Derived" << endl;
    }
};

int main() {
    Derived obj;
    obj.func1();  // Output: Base1
    obj.func2();  // Output: Base2
    obj.func3();  // Output: Derived
    return 0;
}
```

43. Access Modifiers in C++ and Types

Access modifiers control the accessibility of class members. C++ has three access specifiers:

- **Public**: Members are accessible from outside the class.
- **Private**: Members are only accessible within the class.
- **Protected**: Members are accessible within the class and by derived classes.

44. How Public, Private, and Protected Access Modifiers Affect Inheritance

- **Public** members of the base class remain public in the derived class.
- **Private** members of the base class are inaccessible to the derived class.
- **Protected** members of the base class are accessible in derived classes but not outside the class hierarchy.

45. How Access Modifiers Control Member Accessibility in Derived Classes

Public: Inherited public members remain public.
Private: Inherited private members remain private.
Protected: Inherited protected members remain protected.

Example:

cpp
CopyEdit

```
class Base {
public:
    int x;
private:
    int y;
protected:
    int z;
};

class Derived : public Base {
public:
    void show() {
        cout << "Public member: " << x << endl; // Accessible
        // cout << "Private member: " << y << endl; // Error:
inaccessible
        cout << "Protected member: " << z << endl; //
Accessible
    }
```

```
};
```

46. Function Overriding in Inheritance

Function overriding allows a derived class to provide a specific implementation of a function already defined in its base class.

Example:

```
cpp
CopyEdit
class Base {
public:
    virtual void display() {
        cout << "Base display" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {
        cout << "Derived display" << endl;
    }
};
```

47. How to Override a Base Class Function in Derived Class

In the derived class, you simply redefine the function, ensuring it has the same signature as the base class function.

Example:

```
cpp
CopyEdit
class Base {
public:
    virtual void display() {
        cout << "Base class" << endl;
    }
};

class Derived : public Base {
public:
    void display() override {  // Overridden function
        cout << "Derived class" << endl;
    }
};
```

48. The Use of the "Virtual" Keyword in Function Overriding

The `virtual` keyword ensures that the correct function (derived or base) is called based on the actual object type, not the pointer type.

49. Significance of the "Override" Specifier in C++11 and Later

The `override` keyword explicitly indicates that a function is intended to override a base class function. It provides compile-time checking and reduces errors.

Example:

```
cpp
CopyEdit
class Base {
public:
    virtual void display() { cout << "Base display" <<
endl; }
};

class Derived : public Base {
public:
    void display() override { cout << "Derived display" <<
endl; }
};
```

50. Virtual Base Class in C++

A **virtual base class** is used in multiple inheritance to avoid ambiguity when two or more base classes share a common base class.

51. How to Declare and Implement a Virtual Base Class

A base class is declared virtual by prefixing it with the `virtual` keyword.

Example:

```
cpp
CopyEdit
class Base {
public:
    void display() {
        cout << "Base display" << endl;
    }
};
```

```
class Derived1 : virtual public Base {};
class Derived2 : virtual public Base {};

class Final : public Derived1, public Derived2 {};
```

52. Role of Virtual Base Classes in Resolving Ambiguity in Multiple Inheritance

In the case of **multiple inheritance**, if two derived classes inherit the same base class, a **virtual base class** ensures that the base class is only inherited once, avoiding ambiguity.