**Assignment 3( Theory)**

1. What is an object in C++?

Ans An object in C++ is an instance of a class that represents a specific entity with its own data (attributes) and behavior (methods). It occupies memory and is used to access the class's members.

Example: If Car is a class, a specific car like myCar is an object.

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

Ans A class is a user-defined blueprint that defines properties (data members) and behaviors (member functions) for objects. An object is an instance of a class.

| **Aspect** | **Class** | **Object** |
| --- | --- | --- |
| Definition | Blueprint or template | Instance of the class |
| Memory | No memory is allocated | Memory is allocated |
| Declaration | Done once | Can be done multiple times |
| Purpose | Defines properties and actions | Represents a specific real-world entity |

3. Explain the concept of encapsulation with an example.

Ans Encapsulation is the bundling of data and methods that operate on that data within a class, restricting direct access to some components (data hiding). Example:

#include <iostream>

using namespace std;

class BankAccount {

private:

double balance; // Private data

public:

void deposit(double amount) { // Public method

if (amount > 0) balance += amount;

}

double getBalance() { return balance; }

};

int main() {

BankAccount acc;

acc.deposit(1000);

cout << "Balance: " << acc.getBalance() << endl;

// acc.balance = 500; // Error: balance is private

return 0;

}

Here, balance is private and can only be accessed via public methods.

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

Ans A class is defined using the class keyword, followed by the class name and a block containing data members and member functions.

Syntax:

class ClassName {

public:

// Data members and member functions

private:

// Private members

};

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

Ans Syntax:

ClassName objectName; // Static allocation

ClassName\* objectName = new ClassName(); // Dynamic allocation

Example:

Student s1; // Static object

Student\* s2 = new Student(); // Dynamic object

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

Ans Private members in a C++ class are variables or functions that cannot be accessed directly from outside the class. They can only be accessed by the class's member functions or friend functions.

Example

class Car {

private:

int year; // private member

public:

void setYear(int y) {

year = y; // allowed: access within member function

}

int getYear() {

return year;

}

};

Accessing Private Members

You cannot do this:

Car myCar;

myCar.year = 2020; // Error: 'year' is private

Instead, you access them like this using public member functions:

Car myCar;

myCar.setYear(2020);

cout << myCar.getYear();

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

Ans Public members are accessible from anywhere in the program, including outside the class. Access directly using the dot operator (.) for objects or arrow operator (->) for pointers.

Example:

class Student {

public:

string name;

void display() { cout << name << endl; }

};

int main() {

Student s;

s.name = "John"; // Public member access

s.display();

return 0;

}

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

Ans Access specifiers (public, private, protected) control the visibility and accessibility of class members:

Public: Members are accessible from anywhere.

Private: Members are accessible only within the class.

Protected: Members are accessible within the class and in derived classes.

They are significant because they enforce encapsulation, improve security, and provide controlled access to class data.

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

Answer:

#include <iostream>

using namespace std;

class Employee {

private:

double salary; // Private member

public:

string name; // Public member

void setSalary(double s) { // Public method

if (s > 0) salary = s;

}

double getSalary() { return salary; }

};

int main() {

Employee emp;

emp.name = "Alice"; // Public access

emp.setSalary(50000); // Access private member via public method

cout << "Name: " << emp.name << ", Salary: " << emp.getSalary() << endl;

return 0;

}

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

Ans Data hiding is achieved by declaring data members as private or protected, preventing direct access from outside the class. Public methods (getters/setters) are provided to access or modify these members safely. This protects the internal state of the object. As in Question 3 (balance is hidden, accessed via deposit and getBalance).

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

Ans A static data member is a class member that is shared by all objects of the class. It exists independently of any object and is initialized only once. Use to store data common to all instances, e.g., a counter for the number of objects.

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

Ans Declaration: Inside the class with the static keyword. Initialization: Outside the class, typically in the source file.

Syntax:

class ClassName {

public:

static int staticMember; // Declaration

};

// Initialization outside class

int ClassName::staticMember = 0;

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

Ans A static member function is a class function that can be called without creating an object. It can only access static data members or other static functions. Use for operations that don’t depend on object state, e.g., utility functions.

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

Ans

| **Feature** | **Regular Member Function** | **Static Member Function** |
| --- | --- | --- |
| Belongs to | An **object** of the class | The **class itself**, not any specific object |
| Requires object | Yes – called using an object | No – can be called using the class name |
| Access to members | Can access both static and non-static | Can access **only static** members |
| Syntax to call | obj.function() | ClassName::function() |

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

Answer:

#include <iostream>

using namespace std;

class Counter {

public:

static int count; // Static data member

Counter() { count++; } // Increment on object creation

static void displayCount() { // Static function

cout << "Total objects: " << count << endl;

}

};

// Initialize static member

int Counter::count = 0;

int main() {

Counter c1, c2, c3;

Counter::displayCount(); // Call static function

return 0; }

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

Ans A constructor is a special member function that is automatically called when an object is created. It initializes the object’s data members.

Importance:

* Ensures objects are properly initialized.
* Allows setting initial values for data members.
* Supports different ways to create objects (e.g., with or without parameters).

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

Ans The different types of constructors in C++ are:

* Default Constructor: No parameters, initializes members to default values.
* Parameterized Constructor: Takes parameters to initialize members with specific values.
* Copy Constructor: Initializes an object using another object of the same class.

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

Ans A default constructor is a constructor with no parameters. It is used when an object is created without specifying initial values.

Example:

class MyClass {

public:

int x;

MyClass() { x = 0; } // Default constructor

};

int main() {

MyClass obj; // Calls default constructor

return 0;

}

19. How do parameterized constructors work?

Ans Parameterized constructors take arguments to initialize object members with specific values. They are called when an object is created with arguments. Example:

#include <iostream>

using namespace std;

class Student {

public:

string name;

Student(string n) { name = n; } // Parameterized constructor

};

int main() {

Student s("John");

cout << s.name << endl;

return 0;

}

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

Ans A copy constructor creates a new object as a copy of an existing object. It is used during object initialization or passing objects by value.

Syntax:

ClassName(const ClassName& obj);

Example:

#include <iostream>

using namespace std;

class MyClass {

public:

int x;

MyClass(int v) { x = v; }

MyClass(const MyClass& obj) { x = obj.x; } // Copy constructor

};

int main() {

MyClass obj1(10);

MyClass obj2 = obj1; // Copy constructor called

cout << obj2.x << endl;

return 0;

}

21. Explain the concept of constructor overloading.

Ans Constructor overloading allows defining multiple constructors in a class with different parameter lists. The appropriate constructor is called based on the arguments provided. Example:

#include <iostream>

using namespace std;

class Box {

public:

int size;

Box() { size = 0; } // Default constructor

Box(int s) { size = s; } // Parameterized constructor

};

int main() {

Box b1; // Calls default constructor

Box b2(5); // Calls parameterized constructor

cout << b2.size << endl;

return 0;

}

22. How does a constructor initializer list work?

Ans A constructor initializer list initializes data members before the constructor body executes. It is more efficient for initializing const members or objects. Syntax:

ClassName(params) : member1(value1), member2(value2) { /\* body \*/ }

Example:

#include <iostream>

using namespace std;

class MyClass {

int x;

public:

MyClass(int v) : x(v) { // Initializer list

cout << "Constructor called" << endl;

}

int getX() { return x; }

};

int main() {

MyClass obj(10);

cout << obj.getX() << endl;

return 0;

}

23. What is a destructor in C++ and what is its purpose?

Ans A destructor is a special member function called automatically when an object goes out of scope or is deleted. It cleans up resources (e.g., dynamically allocated memory).

Purpose: Prevents memory leaks and ensures proper resource deallocation.

24. How is a destructor declared and defined?

Ans Declaration: Same name as the class, prefixed with ~, no parameters or return type.

Syntax:

~ClassName() { /\* Cleanup code \*/ }

Example:

#include <iostream>

using namespace std;

class MyClass {

public:

MyClass() { cout << "Constructor" << endl; }

~MyClass() { cout << "Destructor" << endl; }

};

int main() {

MyClass obj; // Destructor called when obj goes out of scope

return 0;

}

25. What happens if a destructor is not explicitly defined in a class?

Ans If no destructor is defined, the compiler provides a default destructor that performs no specific cleanup. This is sufficient for classes without dynamically allocated resources but may cause memory leaks if resources need explicit deallocation.

26. Explain the concept of automatic and dynamic storage duration in relation to destructors.

Ans Automatic Storage Duration: Objects declared locally (e.g., MyClass obj;) are destroyed automatically when they go out of scope, triggering the destructor.

Dynamic Storage Duration: Objects created with new (e.g., MyClass\* obj = new MyClass;) persist until explicitly deleted with delete, which calls the destructor. Example:

MyClass obj; // Automatic: Destructor called at end of scope

MyClass\* ptr = new MyClass; // Dynamic: Must call delete ptr

27. How do destructors differ from constructors?

Ans Here the difference:

| **Feature** | **Constructor** | **Destructor** |
| --- | --- | --- |
| Purpose | Initializes an object when it is created | Cleans up when the object is destroyed |
| Name | Same as class name | Same as class name, but with a ~ prefix |
| Parameters | Can have parameters (overloading allowed) | Cannot have parameters (no overloading) |
| Called | Automatically when object is created | Automatically when object goes out of scope |
| Frequency | Called once at creation | Called once at destruction |

28. What is operator overloading in C++ and why is it useful?

Ans Operator overloading allows redefining the behavior of operators (e.g., +, -) for user-defined types (classes).

Why useful:

* Makes code intuitive by allowing natural syntax (e.g., obj1 + obj2).
* Enhances flexibility for custom types.

29. Describe the syntax for overloading an operator.

Ans Syntax:

returnType operator symbol (parameters) { /\* Implementation \*/ }

Example:

class MyClass {

public:

int x;

MyClass operator+(const MyClass& other) { // Overload +

MyClass result;

result.x = x + other.x;

return result;

}

};

30. Which operators can and cannot be overloaded in C++?

Ans Can be overloaded: Arithmetic (+, -), relational (==, <), logical (&&, ||), assignment (=), subscript ([]), function call (()), etc.

Cannot be overloaded: Scope resolution (::), member access (. and .\*), sizeof, typeid, alignof.

Reason: Some operators are tied to language mechanics and cannot be redefined.

31. Provide an example of overloading the "+" operator for a custom class.

Answer:

#include <iostream>

using namespace std;

class Point {

public:

int x, y;

Point(int x = 0, int y = 0) : x(x), y(y) {}

Point operator+(const Point& other) { // Overload +

return Point(x + other.x, y + other.y);

}

void display() { cout << "(" << x << ", " << y << ")" << endl; }

};

int main() {

Point p1(2, 3), p2(4, 5);

Point p3 = p1 + p2; // Calls overloaded +

p3.display(); // Output: (6, 8)

return 0;

}

32. Explain the concept of friend functions in the context of operator overloading.

Ans Friend functions are non-member functions that have access to a class’s private members. They are often used for operator overloading when the operator requires access to private data of two objects (e.g., binary operators like +).

33. What is a friend function in C++ and how is it declared?

Ans A friend function is a non-member function granted access to a class’s private and protected members.

Declaration: Use the friend keyword in the class.

Example:

#include <iostream>

using namespace std;

class MyClass {

private:

int x;

public:

MyClass(int v) : x(v) {}

friend void display(MyClass obj); // Friend function

};

void display(MyClass obj) {

cout << "x = " << obj.x << endl; // Access private member

}

int main() {

MyClass obj(10);

display(obj);

return 0;

}

34. How do friend functions differ from member functions?

Ans Difference Between Friend Functions and Member Functions in C++

| **Feature** | **Member Function** | **Friend Function** |
| --- | --- | --- |
| Belongs to | A specific class | Not a member of the class |
| Access to private data | Yes, directly | Yes, if declared as a friend |
| Syntax | Called using an object: obj.func() | Called like a normal function: func(obj) |
| Declaration | Inside class definition | Declared with friend keyword in class |
| Object parameter | Implicit (this pointer) | Requires object as an argument |

35. Explain the benefits and potential drawbacks of using friend functions.

Ans Benefits:

* Allow non-member functions to access private data, useful for operator overloading or utility functions.
* Enable operations involving multiple classes.

Drawbacks:

* Break encapsulation by exposing private members.
* Can make code less maintainable if overused.
* May lead to tight coupling between classes.

36. What is inheritance in C++ and why is it important?

Ans Inheritance is a mechanism where a class (derived class) inherits properties and behaviors from another class (base class).

Importance:

* Promotes code reusability.
* Supports polymorphism and extensibility.
* Models real-world hierarchical relationships.

37. Explain the different types of inheritance in C++.

Ans Single Inheritance: One base class, one derived class.

Multiple Inheritance: Derived class inherits from multiple base classes.

Multilevel Inheritance: A class is derived from a class that is itself derived.

Hierarchical Inheritance: Multiple classes derived from a single base class.

Hybrid Inheritance: Combination of two or more inheritance types.

38. How do you implement single inheritance in C++?

Ans Use the : operator to specify the base class in the derived class definition. Example:

#include <iostream>

using namespace std;

class Animal {

public:

void eat() { cout << "Eating" << endl; }

};

class Dog : public Animal { // Single inheritance

public:

void bark() { cout << "Barking" << endl; }

};

int main() {

Dog d;

d.eat(); // Inherited from Animal

d.bark();

return 0;

}

39. What is multiple inheritance and how does it differ from single inheritance?

Ans Multiple Inheritance: A derived class inherits from more than one base class.

| **Feature** | **Single Inheritance** | **Multiple Inheritance** |
| --- | --- | --- |
| Definition | A class inherits from **one** base class | A class inherits from **two or more** base classes |
| Syntax | class A : public B | class A : public B, public C |
| Complexity | Simple, easy to manage | More complex, may cause ambiguity (e.g. diamond problem) |
| Use Case | When behavior comes from one base class | When behavior must be combined from multiple sources |

40. Describe hierarchical inheritance with an example.

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

Example:

#include <iostream>

using namespace std;

class Vehicle {

public:

void move() { cout << "Moving" << endl; }

};

class Car : public Vehicle {

public:

void drive() { cout << "Driving car" << endl; }

};

class Bike : public Vehicle {

public:

void ride() { cout << "Riding bike" << endl; }

};

int main() {

Car c;

Bike b;

c.move(); // Inherited

b.move(); // Inherited

return 0;

}

41. What is multilevel inheritance and how is it implemented in C++?

Ans Multilevel inheritance occurs when a class is derived from a class that is itself derived from another class.

Implementation: Use : to chain inheritance.

Example:

#include <iostream>

using namespace std;

class Animal {

public:

void eat() { cout << "Eating" << endl; }

};

class Mammal : public Animal {

public:

void walk() { cout << "Walking" << endl; }

};

class Dog : public Mammal {

public:

void bark() { cout << "Barking" << endl; }

};

int main() {

Dog d;

d.eat(); // From Animal

d.walk(); // From Mammal

d.bark(); // From Dog

return 0;

}

42. Explain the concept of hybrid inheritance.

Ans Hybrid inheritance is a combination of two or more types of inheritance (e.g., multiple and multilevel). It may lead to the diamond problem, which can be resolved using virtual base classes.

Example: A class inheriting from two classes that both inherit from a common base class.

43. What are access modifiers in C++ and what are the different types?

Ans Access modifiers control the accessibility of class members.

Types:

* Public: Accessible from anywhere.
* Private: Accessible only within the class.
* Protected: Accessible within the class and derived classes.

44. How do public, private, and protected access modifiers affect inheritance?

Answer: Public Inheritance: Public members remain public, protected members remain protected, private members are inaccessible.

Protected Inheritance: Public and protected members become protected in the derived class.

Private Inheritance: Public and protected members become private in the derived class.

45. Explain how access modifiers control member accessibility in derived classes.

Ans Public Members: Accessible in derived classes and outside (if public inheritance).

Protected Members: Accessible in derived classes but not outside.

Private Members: Inaccessible in derived classes, regardless of inheritance type. This ensures encapsulation while allowing controlled access in derived classes.

46. What is function overriding in the context of inheritance?

Ans Function overriding occurs when a derived class provides a specific implementation for a virtual function defined in the base class. The derived class function has the same signature (name, parameters, return type). Use: Enables runtime polymorphism.

47. How do you override a base class function in a derived class?

Ans Declare the base class function as virtual and provide a new implementation in the derived class with the same signature.

Example:

#include <iostream>

using namespace std;

class Base {

public:

virtual void show() { cout << "Base" << endl; }

};

class Derived : public Base {

public:

void show() override { cout << "Derived" << endl; }

};

int main() {

Base\* b = new Derived();

b->show(); // Calls Derived::show()

delete b;

return 0;

}

48. Explain the use of the "virtual" keyword in function overriding.

Ans The virtual keyword in a base class function enables runtime polymorphism. It ensures that the function call is resolved based on the actual object type, not the pointer/reference type. Without virtual: The base class function is called (static binding).With virtual: The derived class function is called (dynamic binding).

49. What is the significance of the "override" specifier in C++11 and later?

Ans The override specifier ensures that a derived class function is intended to override a virtual function in the base class. It helps catch errors at compile time if the function does not match a base class virtual function (e.g., wrong signature).

50. What is a virtual base class in C++ and why is it used?

Ans A virtual base class is a base class inherited with the virtual keyword to ensure only one instance of it exists in the derived class hierarchy, avoiding ambiguity in multiple inheritance. It is used resolves the diamond problem by sharing a single copy of the base class.

51. How do you declare and implement a virtual base class?

Ans Use the keyword virtual when inheriting the base class:

class A {

public:

void show() { cout << "Base A" << endl; }

};

class B : virtual public A { };

class C : virtual public A { };

class D : public B, public C {

// Only one copy of A is inherited

};

52. Explain the role of virtual base classes in resolving ambiguity in multiple inheritance.

Ans In multiple inheritance, if two base classes inherit from a common base class, the derived class may end up with multiple copies of the common base (diamond problem). A virtual base class ensures only one instance of the common base class is inherited, eliminating ambiguity when accessing its members.

53. Provide an example of using a virtual base class to avoid the diamond problem in inheritance.

Answer:

#include <iostream>

using namespace std;

class A {

public:

void show() { cout << "Class A" << endl; }

};

class B : virtual public A {}; // Virtual inheritance

class C : virtual public A {}; // Virtual inheritance

class D : public B, public C {};

int main() {

D obj;

obj.show(); // No ambiguity: Single copy of A

return 0;

}

Without virtual, D would have two copies of A, causing ambiguity when calling show().